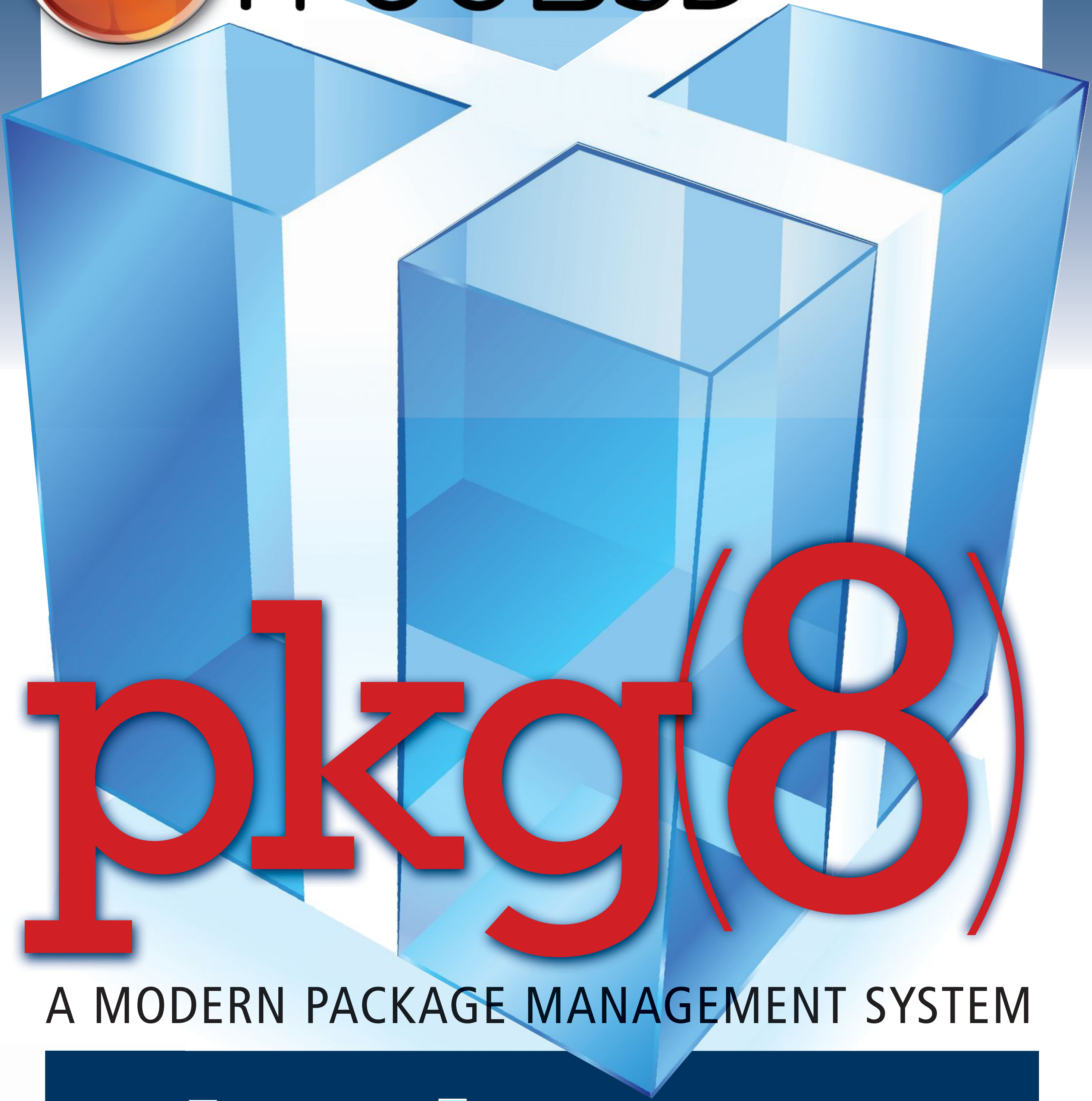
FreeBSD™ JOURNAL

March/April 2014

Vol. 1 • Issue No. 2



pkg(8)

A MODERN PACKAGE MANAGEMENT SYSTEM

- Poudriere
- Journaled Soft-updates
- The Evolution of the PBI Format
- hwpmc(4)

Bulletproof.

Secure your network with Netgate®.

pfSense® Netgate®

7212 McNeil Drive Suite 204 Austin, TX 78729 +1.512.646.4100

Table of Contents

March/April 2014
FreeBSD Journal • Vol. 1, Issue No. 2

Keep FreeBSD Free!
(Donate Today...)

Last year your donations helped FreeBSD by:

- Funding development projects to improve FreeBSD, including: Native iSCSI kernel stack, Updated Intel graphics chipset support, Integration of Newcons, UTF-8 console support, Superpages for ARM architecture, and Layer 2 networking modernization and performance improvements.
- Hiring two more staff members to help with FreeBSD development projects, security, and release engineering.
- Educating the public and promoting FreeBSD. Providing funding and leadership in publishing the FreeBSD Journal. We create high-quality brochures to teach people about FreeBSD. We also visit companies to help facilitate collaboration efforts with the Project.
- Sponsor BSD conferences and summits in Europe, Japan, Canada, and the US.
- Protecting FreeBSD intellectual property and providing legal support to the Project.
- Purchasing hardware to build and improve FreeBSD project infrastructure.

What will your donation help the Foundation accomplish in 2014?

Support FreeBSD by donating to the FreeBSD Foundation

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system. Our mission is to continue and increase our support and funding to keep FreeBSD at the forefront of operating system technology.

Thanks to people like you, the FreeBSD Foundation has been proudly supporting the FreeBSD Project and community for over 13 years now. We are incredibly grateful for all the support we receive from you and so many individuals and organizations that value FreeBSD.

Make a gift to support our work in 2014. Help us continue and increase our support of the FreeBSD Project and community worldwide!

Making a donation is quick and easy. Go to:
www.freebsdfoundation.org/donate

The FreeBSD FOUNDATION

www.freebsdfoundation.org

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher	Walter Andrzejewski walter@FreeBSDjournal.com
Editor-at-Large	James Maurer jmaurer@FreeBSDjournal.com
Art Director	Dianne M. Kischitz dianne@FreeBSDjournal.com
Office Administrator	Cindy DeBeck cindy@FreeBSDjournal.com
Advertising Sales	Walter Andrzejewski walter@FreeBSDjournal.com Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year. (January/February, March/April, May/June, July/August, September/October, November/December)

Published by the FreeBSD Foundation, PO Box 20247, Boulder, CO 80308
ph: 1/720-207-5142 • fax: 1/720-222-2350
e-mail: board@freebsdfoundation.org
Copyright © 2014 by FreeBSD Foundation. All rights reserved.

This magazine may not be reproduced in whole or in part without written permission from the publisher.

More than 1,000 Subscribers Worldwide!

We're excited to announce that the launch of ***FreeBSD Journal*** has been a great success, and, as the title of this letter has already announced, we have had over 1,000 subscribers to date. This is an exciting and auspicious start for a new magazine and we anticipate more growth as the word gets out.

This issue features a set of articles on FreeBSD's new package system. For everyone who has maintained a large installation of FreeBSD systems using the existing ports and package system, the advent of the new system will definitely be a breath of fresh air. The ability to maintain related package sets, build your own repository of packages, and maintain a diverse set of software on top of FreeBSD is a boon to individual users and administrators of large sites alike. Everyone on the mailing lists has been raving about the new package system, and now you can read about how it works directly from the authors themselves—Baptiste Daroussin and Bryan Drewery. PCBSD has had its own packaging system for a while and Kris Moore concludes our packaging coverage with his article on the evolution of the PBI format.

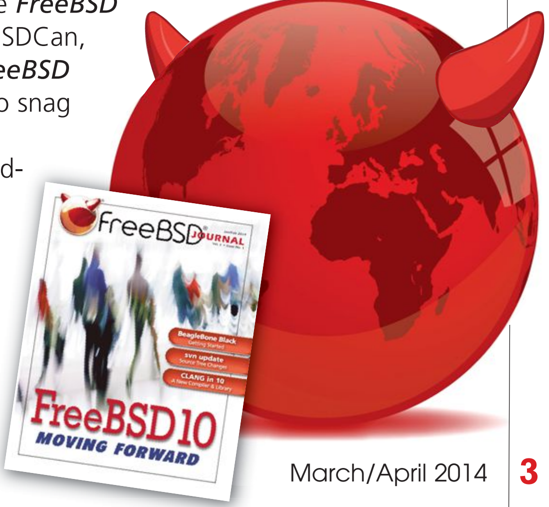
Two articles on timely topics not directly related to packaging round out this issue: the first is on Journaled Soft-updates and the other focuses on performance analysis with the hwpmc(4) device driver and related tools. Finally, when you're looking for an informative "quick read," check out any of this month's columns to keep up-to-date on the latest in the world of FreeBSD.

The Board thanks everyone for their feedback, especially those who sought us out at the recent NYCBSDCon in New York and AsiaBSDCon in Tokyo to discuss this new venture. The response has been gratifying, educational, and appreciated. Readers have already suggested some great ideas for articles and columns in future issues.

Speaking of BSD events, one of our biggest events of the year, BSDCan, is quickly approaching. BSDCan, held in Ottawa, Canada, every May is the largest of the BSD events and attracts developers, vendors, and users of all BSDs for several days of developer summits, tutorials and paper presentations. Representatives of the FreeBSD Foundation and the ***FreeBSD Journal*** will be present at BSDCan, and if you're a reader of ***FreeBSD Journal***, you may be able to snag a collector's item at the Foundation's table—a limited-edition, one-time-only print copy of Issue #1!

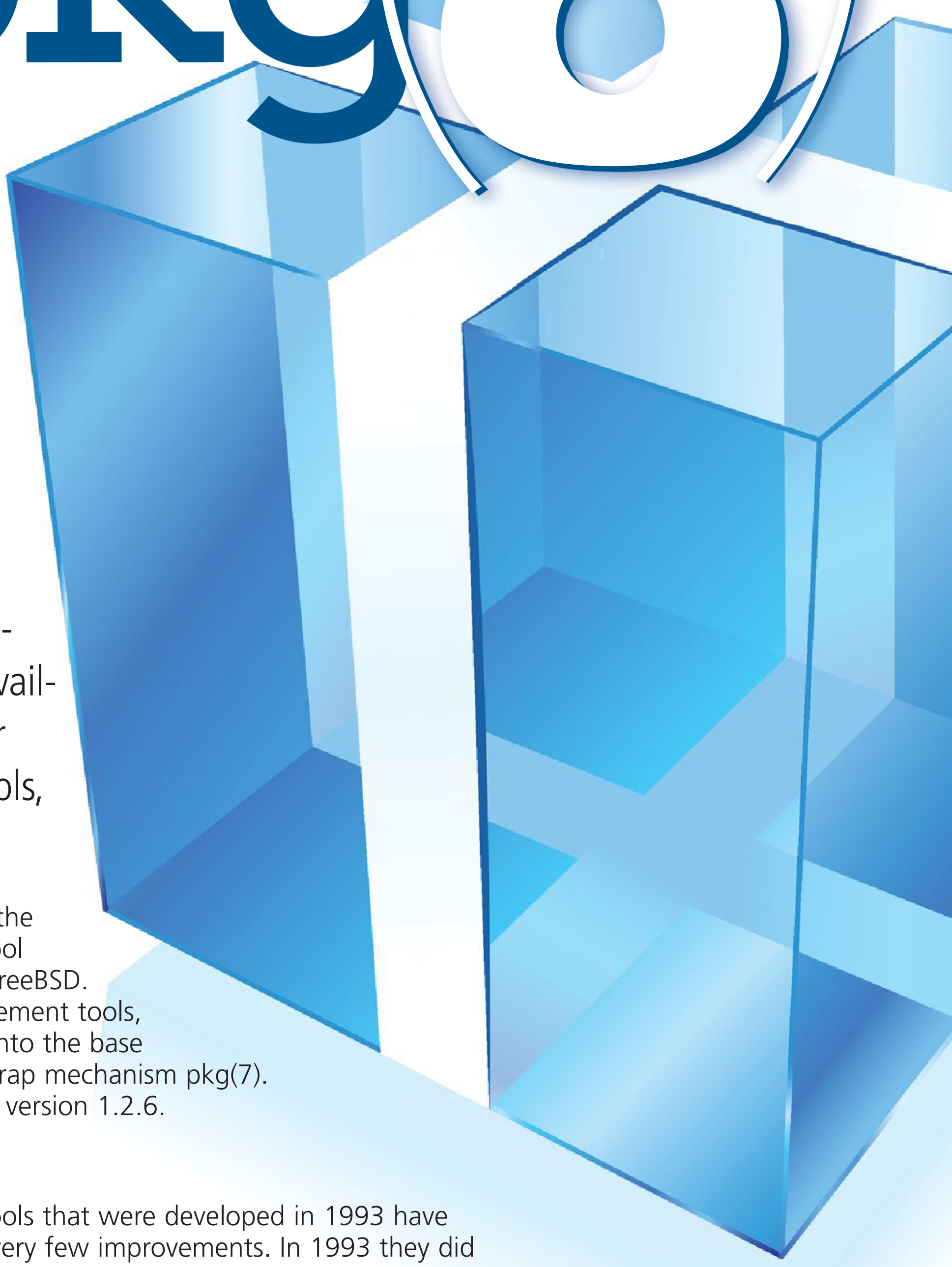
Enjoy!

FreeBSD Journal Editorial Board



By Baptiste Daroussin

pkg(8)



pkg(8) is the new **FreeBSD PACKAGE MANAGER** and the only package manager support for Freebsd 10 and newer. It is available, along with the older package management tools, on FreeBSD 8 and 9.

In September 2014, it will be the only package management tool supported on all versions of FreeBSD. Unlike the older package management tools, pkg(8) is not integrated directly into the base system, instead, there is a bootstrap mechanism pkg(7).

This article covers pkg(8) as of version 1.2.6.

History

The old package management tools that were developed in 1993 have been maintained, but have had very few improvements. In 1993 they did meet the requirements (few packages, simple semantics involved in packaging) for Y2K, but they have become less and less accurate. The design is now not accurate for modern packages and the format itself is weak:

- Missing metadata (license, information, compatible platforms)
- Not able to handle modern package management (impossible to upgrade packages)
- Plist format not able to provide enough control over operations
- Mixed concepts: operations (@exec/@unexec/@chmod...) are mixed

with metadata

- ✕ No repository concept
- ✕ No flexible dependencies (Provides/Requires)
- ✕ No version handling on packages
- ✕ Weak local database
- ✕ Not able to properly track/order dependencies recursively
- ✕ Unsafe: packages can silently overwrite files provided by another package

It was hard to improve as the code base was showing its age, and after spending time trying to improve the old one, we had to face the fact that writing a new tool from scratch would be easier and allow us to start out with a modern package management concept.

Why?

In 2010 Julien Laffaye and I decided to see how complicated it would be to create a package management tool that would be able to properly handle binary upgrades, be fast with a lot of packages installed, be safe, and permit control over the whole processes of package manipulations. We also provided a high-level library, so anyone wanting to write tools that deal with packages would be able to just plug in the library and perform safe operations.

A few years later, pkg(8) has grown, and regular contributors have joined in the development including (but not only): Bryan Drewery, Matthew Seaman, Vsevolod Stakhov.

Basic Principles

pkg(8) is built around some basic principles:

- ★ Operations / metadata separation: All scripts and operations are clearly identified and separated from metadata
- ★ Ports ready: The FreeBSD ports tree contains around 25K ready-to-build packages, a new tool has to handle them with as little change as possible (and make it possible to improve them later)
- ★ A full-featured library: Everything is done in libpkg, a high-level library with a simple API that does all the operations
- ★ Binary package oriented: While transparent for the ports tree and the ports tree tool (the user experience hasn't changed for people using the ports tree directly), pkg(8) is mainly designed to handle binary packages, from installing them, to upgrading them, making sure to not start an operation that is not able to finish successfully, and tracking conflicts ordering dependencies.
- ★ Simplicity: We tried to keep the pkg(8) UI as simple as possible
- ★ Safe defaults: while pkg(8) can provide a variety of options to fit user needs, the dangerous one are off by default.
- ★ Extensible: pkg(8) embeds a plugin mechanism to easily add new features without having to modify pkg(8) itself.

The Bootstrap Mechanism

Preamble

pkg(8) is not available in base and we made a decision to never push it into base for the following reasons:

- ■ FreeBSD does maintain multiple releases at the same time, meaning if you introduce a new feature in pkg(8), we have to wait until the End Of Life of the release not supporting that new feature to be able to use it in the ports. In the long term, this leads to stagnation (no one really improving the package tools) and the introduction of hacks instead of real solutions into the ports tree.
- ■ Requirements in packaging evolve quite quickly and the package tools have to comply with them. The way we did packaging in 1995 and the way we do it now is quite different. Having the ability to improve the package tools very quickly on all supported versions is a major requirement.

The Bootstrap

As pkg(8) is not available directly from the base system, a bootstrap mechanism pkg(7) has been introduced into the base system and designed to be seamless.

FreeBSD 8.4 is the oldest version that includes the bootstrap tool. FreeBSD 10 is the only version to have it in its final version.

On FreeBSD 10, pkg(7) is, in fact, /usr/sbin/pkg. When running it will look for the pkg(8) binary (by default in ${LOCALBASE}/sbin). If the binary has been found, then it will directly execute it. If not found, it will suggest that the user to bootstrap it.

Bootstrapping consists of fetching the latest pkg(8) packages from the official FreeBSD repositories (http://pkg.FreeBSD.org) for a given

version on a given architecture.
- ★ It will also fetch a signature file (pkg.txz.sig) containing a signature and a public key
- ★ Verify the public key against a list of fingerprints of known trusted public keys
- ★ Validate that the fetched pkg(8) package matches the signature
- ★ Extract pkg-static from the package

Install the package with the extracted pkg-static
- ★ Execute pkg(8) with the arguments passed to the bootstrap

To determine where to fetch the package and how to validate it, pkg(7) shares the exact same mechanism of configuration files for repositories (which will be described later).

pkg(8) Configuration Option

In pkg(8) almost all options can be overwritten by an environment variable.
The command `pkg -vv` will show all the support options and the current setup:

```
Version                    : 1.2.6
PACKAGESITE                :
PKG_DBDIR                  : /var/db/pkg
PKG_CACHEDIR               : /var/cache/pkg
PORTSDIR                   : /usr/ports
PUBKEY                     :
HANDLE_RC_SCRIPTS          : no
ASSUME_ALWAYS_YES          : no
REPOS_DIR                  : [
  /etc/pkg/,
  /usr/local/etc/pkg/repos/,
]
PLIST_KEYWORDS_DIR         :
SYSLOG                     : yes
AUTODEPS                   : yes
ABI                        : freebsd:11:x86:64
DEVELOPER_MODE             : no
PORTAUDIT_SITE             : http://portaudit.FreeBSD.org/auditfile.tbz
VULNXML_SITE               : http://www.vuxml.org/freebsd/vuln.xml.bz2
MIRROR_TYPE                : SRV
FETCH_RETRY                : 3
PKG_PLUGINS_DIR            : /usr/local/lib/pkg/
PKG_ENABLE_PLUGINS         : yes
PLUGINS                    : [
]
DEBUG_SCRIPTS              : no
PLUGINS_CONF_DIR           : /usr/local/etc/pkg/
PERMISSIVE                 : no
REPO_AUTOUPDATE            : yes
NAMESERVER                 :
EVENT_PIPE                 :
FETCH_TIMEOUT              : 30
UNSET_TIMESTAMP            : no
SSH_RESTRICT_DIR           :
PKG_SSH_ARGS               :
PKG_ENV                    : {
}
DISABLE_MTREE              : no
DEBUG_LEVEL                : 0
ALIAS                      : {
}
```

(continues next page)

```
Repositories:
    FreeBSD: {
    url                 : "pkg+http://pkg.FreeBSD.org/freebsd:11:x86:64/latest",
    enabled             : yes,
    mirror_type         : "SRV",
    signature_type      : "FINGERPRINTS",
    fingerprints        : "/usr/share/keys/pkg"
  }
```

- PKG_DBDIR is the directory where the database for local and remote packages is located
- PKG_CACHEDIR is the directory from which the packages will be fetched before being installed
- HANDLE_RC_SCRIPT will tell pkg(8) to automatically restart the rc script after an upgrade
- SYSLOG will make pkg(8) log everything it is doing via the syslog mechanism
- REPOS_DIR is the directories where pkg can find the binary configurations files (when set from environment, it should be defined the same way PATH is defined: REPOS_DIR="/path1:/path2:/path3"
- PLUGINS_CONF_DIR is the directory where the plugins can find their configuration files

pkg(8) Cold Start

For the rest of this article we will consider a setup where LOCALBASE is defined as /usr/local.

Important files or directories around pkg(8):

```
/etc/pkg/*.conf
/usr/local/etc/pkg.conf
/usr/local/etc/pkg/repos/*.conf
/usr/local/lib/libpkg.so.1
/usr/local/sbin/pkg
/usr/local/sbin/pkg-static
```

pkg-static is a statically linked version of pkg(8) that allows users to do binary package manipulation, even when upgraded from a major version to another major version of FreeBSD.

When the pkg(8) binary is invoked, it will first load /usr/local/etc/pkg.conf unless another location is specified via the -C option. For any configuration entry, if there is already an environment variable defining it, it will have priority over the configuration file.

Once the configuration file has been read, pkg(8) can look for repository definitions in the directory defined via the REPOS_DIR. It will read all the .conf files in those directories and will look for 1 or N entries like this:

```
<NAME> : {
      url : "<url>",
      enabled: true,
      mirror_type: "SRV",
      signature_type: "FINGERPRINTS",
      fingerprints:
"/usr/share/keys/pkg",
      pubkey: "/etc/pkg/pub.key"
}
```

First time <NAME> is found the "url" element is mandatory. Any time after that, the options it defines will overwrite the previously defined one.

- **url**: List the url from which pkg(8) should fetch the packages. ssh, http(s), ftp, file are supported protocols. If the MIRROR_TYPE is set to "SRV", then the url should be preceded by "pkg+"
- **enabled**: is always true by default, one can set it to false to disable a given repository
- **mirror_type**: by default it is NONE. Possible values are:
  HTTP: The repository URL should download a text document containing a sequence of lines beginning with `URL:' followed by any amount of while space and one URL for a repository mirror. Any lines not matching this pattern are ignored. Mirrors are tried in the order listed until a download succeeds.
  SRV: For an SRV-mirrored repository, where the URL is specified as pkg+http://pkgrepo.example.org/ , SRV records should provide the list of mirrors where the SRV priority and weight parameters are used to control search order and traffic weighting between sites, and where the port number and hostname are used to construct the individual mirror URLs.
- **signature_type**: by default it is NONE, and the following entries can be set here:
  FINGERPRINTS: the public key will be fetched along with the catalog and verified against a list of revoked and trusted fingerprints of public keys
  PUBKEY: use a public key on the system to validate the remote catalog
- **fingerprints**: path to a directory should contain two subdirectories: "trusted" and "revoked", each of which should contain files describing the fingerprint of the concerned public key and the function used to create that fingerprint (right now only sha256 is supported as a function). This entry is only useful if "signature_type" is set to "FINGERPRINTS"
- **pubkey**: path to the public key is only useful if "signature_type" is set to "pubkey"

Different Commands

Unlike the old package management tools, pkg is a single binary with a lot of small commands:

```
$ pkg help
Usage: pkg [-v] [-d] [-l] [-N]
       [-j <jail name or id>|-c <chroot path>]
       [-C<configuration file>] [-R <repo config dir>]
       <command> [<args>]

Global options supported:
      -d          Increment debug level
      -j          Execute pkg(8) inside a jail(8)
      -c          Execute pkg(8) inside a chroot(8)
      -C          Use the specified configuration file
      -R          Directory to search for individual repository configurations
      -l          List available commands and exit
      -v          Display pkg(8) version
      -N          Test if pkg(8) is activated and avoid auto-activation


Commands supported:
      add         Registers a package and installs it on the system
      annotate    Add, modify or delete tag-value style annotations on packages
      audit       Reports vulnerable packages
      autoremove  Removes orphan packages
      backup      Backs-up and restores the local package database
      check       Checks for missing dependencies and database consistency
      clean       Cleans old packages from the cache
      config      Display the value of the configuration options
      convert     Convert database from/to pkgng
      create      Creates software package distributions
      delete      Deletes packages from the database and the system
      fetch       Fetches packages from a remote repository
      help        Displays help information
      info        Displays information about installed packages
      install     Installs packages from remote package repositories
      lock        Locks package against modifications or deletion
      plugins     Manages plugins and displays information about plugins
      query       Queries information about installed packages
      register    Registers a package into the local database
      remove      Deletes packages from the database and the system
      repo        Creates a package repository catalogue
      rquery      Queries information in repository catalogues
      search      Performs a search of package repository catalogues
      set         Modifies information about packages in the local database
      ssh         ssh packages to be used via ssh
      shell       Opens a debug shell
      shlib       Displays which packages link against a specific shared library
      stats       Displays package database statistics
      unlock      Unlocks a package, allowing modification or deletion
      update      Updates package repository catalogues
      updating    Displays UPDATING information for a package
      upgrade     Performs upgrades of packaged software distributions
      version     Displays the versions of installed packages
      which       Displays which package installed a specific file
```

(continues next page)

To keep the system up to date, a normal user will just have to run a couple of commands:

```
$ pkg upgrade
$ pkg autoremove
```

The first will upgrade all the installed packages to their latest version, the second will remove the (now) orphans that are no longer depended upon and were not purposely installed by the user.

Creating a Package Outside of the Ports Tree

While the ports tree is the natural way of creating packages for FreeBSD, it is easy to create your packages outside of the ports tree.

First let's have a look at what `pkg create` expects:

```
Usage: pkg create [-On] [-f format] [-o outdir] [-p plist] [-r rootdir] -m manifestdir
       pkg create [-Ognx] [-f format] [-o outdir] [-r rootdir] pkg-name ...
       pkg create [-On] [-f format] [-o outdir] [-r rootdir] -a
```

The first line is what we are looking for. We assume that the sources have been built and installed in a given DESTDIR and we want to package `/usr/local/bin/foo` and `/usr/local/lib/libbar.so.1`

```
$ find ${DESTDIR}
${DESTDIR}/usr/local/bin/foo
${DESTDIR}/usr/local/lib/libbar.so
${DESTDIR}/usr/local/lib/libbar.so.1
```

First, we need to create a directory into which we can push the manifest:

```
$ mkdir ${MANIFESTDIR}
```

Next create a ${MANIFESTDIR}/+MANIFEST

```
name: foo
version: 1.0
origin: mycompany/foo
categories: [ mycompany, devel ]
comment: foo is a nice tool
www: http://mycompany/foo
maintainer: me@mycompany
prefix: /usr/local
```

This is the minimum necessary and is written in libucl format, which is very flexible and allows different syntaxes. The description for the packages can be added via a file `${MANIFESTDIR}/+DESC` or by adding the following lines to the manifest:

```
desc: <<EOD
foo is a fantastic tool developed by my company
on top of libbar.
EOD
```

A message can be added to the package by either writing it in a ${MANIFESTDIR}/+DISPLAY, or by adding the following line to the manifest:

```
message: <<EOD
This is a message for our user, I hope you do like foo
EOD
```

Create a plist file (as the ports do):

```
bin/foo
lib/libbar.so.1
```

The package can now be created:

```
$ pkg create -m ${MANIFESTDIR} -p plist -r ${DESTDIR} -o ${OUTDIR}
```

This will create a `${OUTDIR}/foo-1.0.txz` ready to be distributed.

The best way to distribute it is to create your own repository:

```
$ pkg repo ${OUTDIR}
```

Focus on Some Features

Access Method

pkg(8) supports a couple different transport mechanisms: - http - ftp - file - ssh (this requires the remote server to also have pkg(8) installed)

In the design of pkg, it was determined that the list of mirrors should not be on the end-user side, but rather dynamically discovered, therefore, two different mechanisms are supported to allow retrieval of the list of mirrors:

- HTTP: The repository URL should download a text document containing a sequence of lines beginning with `URL:' followed by any amount of while space and one URL for a repository mirror. Any lines not matching this pattern are ignored. Mirrors are tried in the order listed until a download succeeds.
- SRV: For an SRV-mirrored repository where the URL is specified as http://pkgrepo.example.org/SRV, records should be set up in the DNS:

```
    $ORIGIN example.com

_   http._tcp.pkgrepo IN SRV 10 1 80 mirror0

                      IN SRV 20 1 80 mirror1
```

where the SRV priority and weight parameters are used to control search order and traffic weighting between sites, and the port number and hostname are used to construct the individual mirror URLs.

Because we fetch packages from the Internet, it is important to make sure the packages are the ones you really expect to receive. Unlike most package systems, pkg(8) does not sign individual packages, but rather the "catalogue" which contains checksums for every single package available. Two mechanisms are provided:

- FINGERPRINTS: pkg will fetch the public certificate along with the catalogue and the signature of the catalogue. It will check the fingerprint of the certificate against a local list of trusted and revoked signatures. If the signature is considered as trusted, then the catalogue will be verified against the signature using the trusted certificate
- PUBKEY: pkg will expect to find a local public certificate and to have a signature with the catalogue. It will check the catalogue against the signature using that certificate.

Orphan Tracking

While a system is live, lots of packages get installed and uninstalled along with their dependencies. pkg(8) tracks what has been explicitly installed by a user and what has been installed automatically as a dependency.

Being able to track gives pkg(8) the ability to provide the "autoremove" sub command:

```
$ pkg autoremove
```

It will suggest removing all the packages that have been installed as dependencies and are not needed anymore. The ports tree has been modified to provide this information when installing a package, meaning that after installing a package with a lot of build dependencies, "pkg autoremove" will suggest removing all the build-only dependencies.

It is possible to mark a package as automatically installed with the following command:

```
$ pkg set -A 1 mypkg
```

Or to unmark it:

```
$ pkg set -A 0 mypkg
```

Multi Repository

pkg(8) is focused on binary packages as a primary target, consequently it supports a repository concept which consist of a bunch of packages and a catalogue containing all the metadata about those packages.

It supports multiple repositories, for example, the official FreeBSD repository, along with a custom repository. A simple use case would be when a user needs the apache module for php, he cannot use the official repositories, because the default php package does not provide such a module. It is possible in that case to simply build the php packages using a building tool like poudriere and create a repository with the custom php packages in it, and use the FreeBSD repository for all other packages:

```
$ pkg install -r myrepo php
```

The above command will install the php package enforcing the installation from the "myrepo" repository.

```
$ pkg annotate -A php repository myrepo
```

will make sure that during the regular "pkg upgrade" in the future, pkg(8) will only check "myrepo" for updates concerning php.

Lock/Unlock

Keeping a given version of a package can sometimes be critical for a system, but upgrading all other packages is then required (security fixes for example). pkg(8) provides a subcommand to allow upgrading the system without upgrading given packages:

```
$ pkg lock mysql-server
```

pkg unlock will allow removal of that lock.

Security

FreeBSD used to provide a third-party tool (portaudit) to audit the installed package database against the list of known vulnerabilities provided via vuxml. In pkg(8), we decided this important feature should be part of the package management tool:

```
$ pkg audit
```

will do such reports.

Powerful Queries

pkg(8) provides two interfaces (query and rquery) to help gather information about locally installed packages and remotely available packages. Both interfaces make it possible to create powerful custom queries on the database and also to control its output.

This makes it possible to simplify integration with scripts:

```
#!/bin/sh


eval `pkg query "WWW=%w ; NAME="%n" ; VERSION="%v" fossil `

echo "${NAME} is installed at version ${VERSION} and its website is: ${WWW}"

# can be also done that way

pkg query "%n is install at version %v and its website is: %w"

#!/bin/sh

eval `pkg query "W=%w; N="%n"; V="%v" fossil `

echo "${N} from ${W}, installed at version ${V}"

# can be also done that way

pkg query "%n from %v, installed at version %v" fossil
```

This can be combined with the pkg(8) alias mechanism to easily provide a new subcommand to create a command that will list all packages explicitly installed by the user. Add the following to pkg.conf:

```
alias: {

    explicit: query -e "%a == 0" "%n-%v"
}
```

now the command pkg exists and is presenting all the packages explicitly installed in the form of one line per package with <name>-<version>.

Future Challenges of pkg

With the next version, we plan to include a lot of new features and improve the user experience. pkg(8) has recently gained a
real sat, which is the foundation for improving both the flexibility of the package format and the package building tools (the ports tree).

By permitting smart dependencies instead of depending on the specific version of a package, we can depend on a valid range of versions.

Allow the inclusion of provides/requires: do not depend on a package, but rather on a feature (Requires: perl, http). Provides/Requires can be very useful in multiple cases, for example:

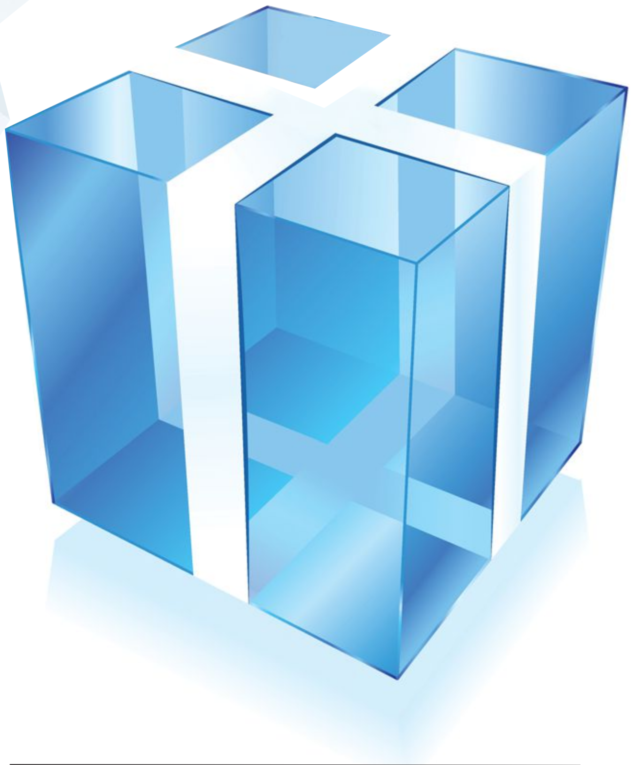
- program A requires libMagick.so and ImageMagick and ImageMagick-nox11 both provide libMagick.so. That permits the user to select the version he wants instead of havingto rebuild the A package and choose which ImageMagick package to use at build time.
- progam B depends on a perl interpreter, it can just requires: perl and not a specific version of perl. Right now the version is hard--coded in the dependency.

Work has also begun on having a pluggable backend for pkg(8) with the current one (the binary repositories) being just one of them. One can also imagine a "ports tree" backend, a CPAN,pear,pypy,gem backend.

For now in the background, pkg(8) uses origin (aka category/portname) to identify a package (determining what is an upgrade of what and so on). This is due to the fact that the same port could have multiple different names in the ports tree depending on options. Or, multiple ports could have the same name but different versions (and still remain different ports and not two versions of the same port). In the meantime, we have been able to fix the ports so that package names are consistent again, meaning we can switch back pkgname as the identifier for packages.

This change, while it doesn't sound very significant, opens a lot of new possibilities for the ports tree:

- sub packages: from one source in one port build multiple packages (like having one single port for php and all its modules)
- flavors (provides): the same package is built with different options. ●



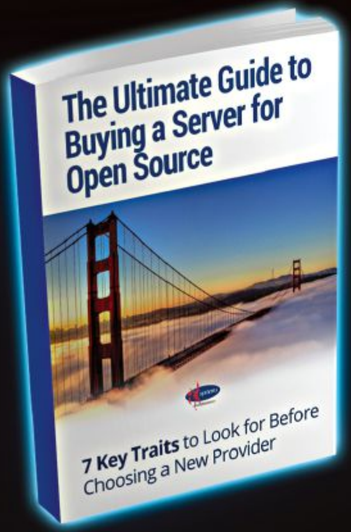
Baptise Daroussin is a UNIX system engineer and resides in France. He has been a FreeBSD user since 1999, a port committer since 2010, a source committer since 2011 and a member of the port management team since 2011. He is the author of pkg(8), and is responsible for important recent changes in the ports tree (new options framework, USES, lots of refactoring of the infrastructure). He is also the author of Poudriere, a package building and testing tool.

Certified FreeBSD Servers...*Actually* Built By Champions Of FreeBSD

"iXsystems Takes The Headache Out Of Server Shopping."

Allan Jude - *www.scaleengine.com*

Savvy FreeBSD pros like Allan Jude know there is a lot more to choosing a server provider than finding the cheapest quote. If your company is expanding, starting a new project, or looking to upgrade hardware then check out these tips to avoid some of the common pitfalls of buying new hardware.


The Ultimate Guide to Buying a Server for Open Source
7 Key Traits to Look for Before Choosing a New Provider

What To Look For

1. Free Pre-Purchase Consultation

The test of a good server provider starts before the first sales proposal arrives in your inbox. You'll know they want more than a sale when they spend more time figuring out what you actually need instead of forcing a cookie-cutter solution.

2. Servers Specifically Built For FreeBSD

The big name vendors are not experts in FreeBSD and that is a FACT. Instead, look for companies that actually have FreeBSD experts in-house. This will ensure you get a server that is finely tuned for FreeBSD.

To learn about the 7 key traits you must absolutely demand from your server provider, go to iXsystems.com/FreeBSDJournal/ and grab your free copy of **The Ultimate Guide To Buying A Server For Open Source.**

Featuring Intel® Xeon® Processors

Call iXsystems toll free today! **1-855-GREP-4-IX**

Stop using portmaster, portupgrade and ports on your servers and switch to packages.

Poudriere

BY BRYAN DREWERY

Setting up your own package builds with Poudriere takes only a few minutes and will save you a lot of time in the future.

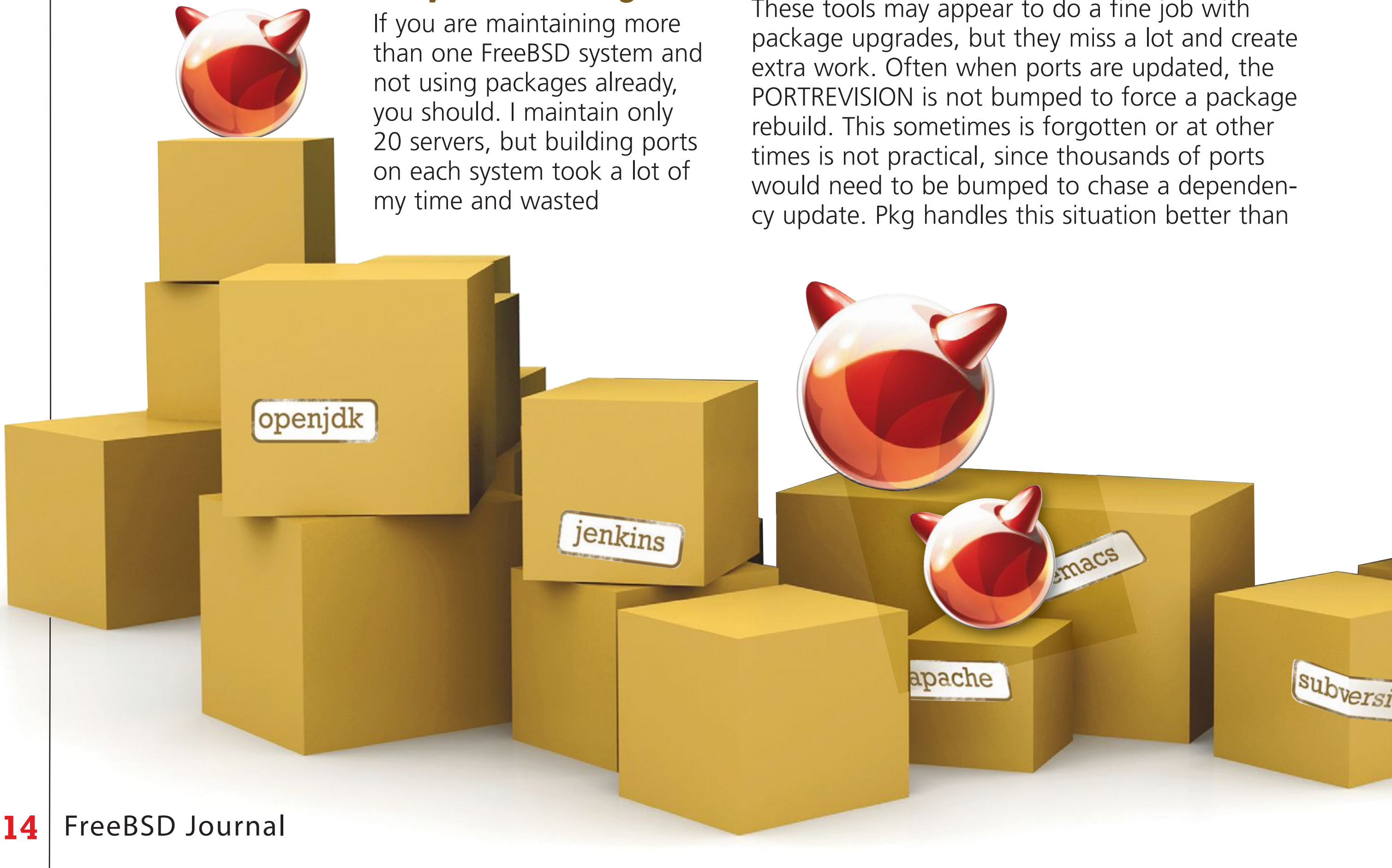
Since November 2013, FreeBSD has provided official packages for Pkg, formerly known as pkgng. The 10 release also brought the first signed packages. The project uses Poudriere for package building.

Why Use Packages

If you are maintaining more than one FreeBSD system and not using packages already, you should. I maintain only 20 servers, but building ports on each system took a lot of my time and wasted resources on production machines.

When building ports on multiple servers, it is very easy to get their options or versions out of sync. By building packages once on one system, I lessened the load on my systems, lessened the amount of work I had to do and made all my systems consistent. Instead of dealing with the same failure on each system, I only need to handle it on the build system.

Until Pkg was available, I never really considered using packages. The old style pkg_install packages were fine for initial system installation, but there was no built-in way to upgrade, except to remove all and install a new set. You had to use a tool such as portmaster or portupgrade and have an INDEX or a ports tree checked out. These tools may appear to do a fine job with package upgrades, but they miss a lot and create extra work. Often when ports are updated, the PORTREVISION is not bumped to force a package rebuild. This sometimes is forgotten or at other times is not practical, since thousands of ports would need to be bumped to chase a dependency update. Pkg handles this situation better than



the old system. Pkg can also detect when the selected options for installed packages have changed from the available remote packages and will reinstall them automatically. The old tools would require recursively reinstalling packages sometimes and not others. There was too much manual work involved with the old package system. The goal of Pkg is to have a built-in upgrade process that removes manual intervention. There is still some work to do on removing some of the manual intervention, but it is already far better than the old system.

Custom Package Options

Why would you need to deviate from the official packages? The ports framework provides options support for ports to change build-time configuration. Not all applications support runtime configuration. Some applications must be compiled differently depending on which features are enabled. Others have options simply to lessen the amount of features and dependencies in the default port. For server administrators, this can quickly lead to finding that some of the default packages do not meet their requirements. One common example is that PHP comes in CGI mode by default without any support for apache+mod_php or the more flexible PHP-FPM. Another common issue with the default packages is that they come with X11 support which may be undesirable on non-desktop environments. Perhaps you have custom ports or custom patches for some ports. By building your own packages you regain control over which options packages are built with and how often updates are available.

Some other reasons to build your own packages are when you are dealing with restrictive licenses for which the FreeBSD project is unable to ship packages or if your system is highly customized and not ABI compatible with FreeBSD.

There are a few ways to get custom packages. Pkg supports using multiple repositories. It can be set up to use the official FreeBSD repository as a primary and a custom one as a secondary. Pkg is not limited by the number of repositories it can track and they can be reordered for priority. The problem with multiple repositories is that it can currently be difficult to maintain. When Pkg detects that an installed package has different options or dependencies from a repository it is tracking, the package will be reinstalled from potentially any remote version. You can either lock the package during upgrades with `pkg lock PKGNAME` and `pkg unlock PKGNAME` or bind it to a specific repository with `pkg annotate -A PKGNAME repository REPONAME`. There is also the subtle problem of keeping the ports tree for your custom repository in sync with the FreeBSD packages. Since packages are built from a ports tree snapshot taken once a week, if your custom repository does not match it may lead to conflicts. It is much simpler to just build an entire package set of just what you need with the options that you want. On your systems you would only track your one repository and not include the FreeBSD one. This also has the benefit of using your own infrastructure for distributing packages which can speed up upgrades substantially.

Building Packages

For the longest time Tinderbox was the popular go-to tool for building packages. Other people would just install all ports on one system and then create packages from that system and copy them to other systems. This method is not recommended because the packages are created in an unclean environment that is constantly growing larger and more polluted. Even using portmaster with ports today and creating Pkg packages from those for distribution is not recommended for the same reasons. It is better to use a system designed for creating package sets.

Poudriere (roughly pronounced poo-dree-year, French for "powder keg") was written as a faster and simpler replacement for Tinderbox. It was written by the Pkg author Baptiste Daroussin and is now mostly maintained by me along with Baptiste and some other contributors. It has quickly become the de-facto FreeBSD port testing and package building tool. It is the

official build cluster tool and is also used by the FreeBSD Ports project for testing sweeping patches in what are called "exp-runs". It is written in POSIX shell and is slowly being moved to C components. Unlike Tinderbox, it has no dependencies and does not require a database. It has been greatly optimized to be highly parallel in all operations. It uses jails to build ports in sandboxed environments in very strict conditions. Jail creation is done once with a simple command. During builds, the jail is cloned automatically for each CPU being used to give ports a clean place to build. Builds can occur on UFS, ZFS or the TMPFS file systems. UFS is high I/O, low RAM, slow build and TMPFS is low I/O, high RAM and very fast build. It is also configurable such that only some parts of the build use TMPFS while others use UFS/ZFS to allow some compromise on lower memory machines. An amd64 host can also build i386 packages with no extra effort. Packages can be built for the current host version or older. For example, if the host machine is 9.2, it can build 9.2, 9.1 and 8.3 package sets.

Poudriere does incremental builds by default to only rebuild what is needed. The incremental build checks for changed options, missing dependencies, changed dependencies, new versions, and changed pkgnames. If any of those changed it will rebuild that port. This also causes anything depending on that port to be rebuilt. This is sometimes overkill, but ensures that no port change is missed in package creation. There is also built-in ccache support which can help port rebuilding time when dependencies change. Build times of package sets vary, but on a system with multiple CPU and enough RAM, a few hundred ports can typically build in an hour or two.

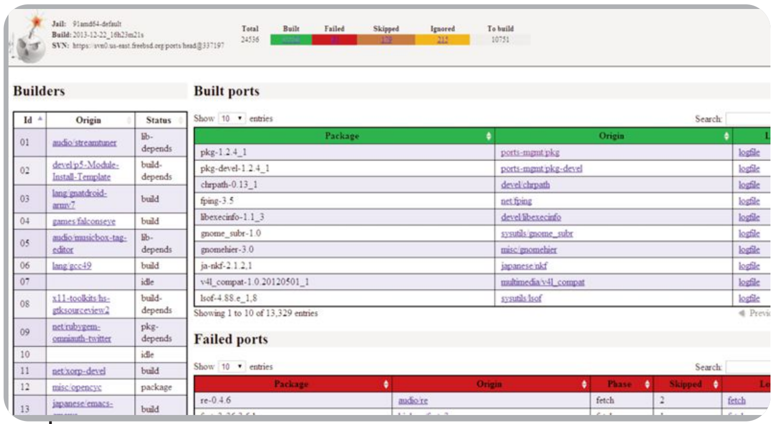
Poudriere has a read-only, real-time web interface that allows monitoring the status of builds. This interface does not require any server-side CGI or scripting support, as Poudriere just writes out a status file in JSON and then web interface uses it. It is not as nice as the Tinderbox interface, but there are plans to improve it more in the future. The 3.1 version has been incrementally improved to be more responsive and allow searching and sorting each sub-list of packages. See screen shot.

Poudriere also has a feature called a "set". This allows having multiple saved options, `make.conf` files and resulting package sets for each named "set". This removes the need to have multiple jails for the same target version/architecture. For example, this can be used to create a PHP 5.3 package set named "php53" and a PHP 5.5 package set named "php55" using one jail on the build system. When building the set would be specified with "-z setname", i.e. "`bulk –a php53 –j 91amd64`" would produce packages in `/usr/local/poudriere/data/packages/91amd64-default-php53`. The "default" refers to the ports tree, which can also be changed with the "-p" option.

The upcoming Poudriere 3.1 release also brings some interesting new features. One of the major ones is named ATOMIC_PACKAGE_REPOSITORY. It prevents the repository from being modified until a build is completed. Currently in 3.0 the repository has packages deleted at startup and packages being modified during build, thus disallowing serving it directly over http. This is enabled by default. It works by hard-link copying the package directory into a .building directory during startup, then when the build completes the .building directory is renamed to a .real_TIMESTAMP directory and the top-level .latest symlink in the repository is updated to point to the new build. There are still potential problems with changing the repository during a pkg upgrade job, but the window for problems is far smaller than without this (Box 1).

This feature also allows doing dry-runs with bulk to see what would be done by using `bulk –n`.

Atomic package repository also allows keeping old package sets. This is not enabled by default but can be enabled by setting KEEP_OLD_PACKAGES and KEEP_OLD_PACKAGES_COUNT. By default, 5 sets are kept. With this you could rollback a system by changing the .latest symlink to an old set and then running `pkg upgrade –f` on a server to force it to


Jail: 91amd64-default
Build: 2013-12-22_16h23m21s
SVN: https://svn0.us-east.freebsd.org/ports/head@337197

Total	Built	Failed	Skipped	Ignored	To build
24536	[illegible]	[illegible]	179	212	10751

Builders

Id	Origin	Status
01	audio/streamtuner	lib-depends
02	devel/p5-Module-Install-Template	build-depends
03	lang/gnatdroid-armv7	build
04	games/falconseye	build
05	audio/musicbox-tag-editor	lib-depends
06	lang/gcc49	build
07		idle
08	x11-toolkits/hs-gtksourceview2	build-depends
09	net/rubygem-omniauth-twitter	pkg-depends
10		idle
11	net/xorp-devel	build
12	misc/opencyc	package
13	japanese/emacs-	build

Built ports

Show 10 entries Search:

Package	Origin	L
pkg-1.2.4_1	ports-mgmt/pkg	logfile
pkg-devel-1.2.4_1	ports-mgmt/pkg-devel	logfile
chrpath-0.13_1	devel/chrpath	logfile
fping-3.5	net/fping	logfile
libexecinfo-1.1_3	devel/libexecinfo	logfile
gnome_subr-1.0	sysutils/gnome_subr	logfile
gnomehier-3.0	misc/gnomehier	logfile
ja-nkf-2.1.2,1	japanese/nkf	logfile
v4l_compat-1.0.20120501_1	multimedia/v4l_compat	logfile
lsof-4.88.e_1,8	sysutils/lsof	logfile

Showing 1 to 10 of 13,329 entries

Failed ports

Show 10 entries Search:

Package	Origin	Phase	Skipped	Lo
re-0.4.6	audio/re	fetch	2	fetch

Poudriere 3.1 Web Interface Preview

Box 1.
Atomic package
repository layout

```
/usr/packages/exp-91amd64-commit-test # ls -al
total 13
drwxr-xr-x   7 root  wheel  12 Mar  2 01:59 ./
drwxr-x—x  26 root  wheel  32 Mar  2 01:13 ../
lrwxr-xr-x   1 root  wheel  16 Mar  2 01:59 .latest@ -> .real_1393747164
drwxr-xr-x   4 root  wheel   7 Mar  1 16:59 .real_1393714735/
drwxr-xr-x   4 root  wheel   7 Mar  2 00:40 .real_1393742366/
drwxr-xr-x   4 root  wheel   7 Mar  2 00:58 .real_1393743542/
drwxr-xr-x   4 root  wheel   7 Mar  2 01:05 .real_1393743901/
drwxr-xr-x   4 root  wheel   7 Mar  2 02:00 .real_1393747164/
lrwxr-xr-x   1 root  wheel  11 Nov 19 17:20 All@ -> .latest/All
lrwxr-xr-x   1 root  wheel  14 Nov 19 17:20 Latest@ -> .latest/Latest
lrwxr-xr-x   1 root  wheel  19 Nov 19 17:20 digests.txz@ -> .latest/digests.txz
lrwxr-xr-x   1 root  wheel  23 Nov 19 17:20 packagesite.txz@ -> .latest/packagesite.txz
```

reinstall all packages from the remote repository. This would downgrade all to the old set.

Another upcoming feature for 3.1 is named poudriered. It will allow non-root usage of poudriere through a socket to a root daemon. This will allow queueing jobs as well as queueing a job for all jails. It is configurable in a similar way as sudo to be able to restrict subcommands and even arguments to specific users and groups. More improvements, such as daemon privilege separation, are planned for 3.2/4.0.

Setup and usage of poudriere is simple and fast. Install poudriere, create a jail, checkout a ports tree, create a file with a list of ports, optionally create a private/public keypair for a

Box 2.
Typical
Poudriere setup

```
$ pkg install ports-mgmt/poudriere
$ cp /usr/local/etc/poudriere.conf.sample /usr/local/etc/poudriere.conf
# Modify configuration.
$ vim /usr/local/etc/poudriere.conf
# Create a ports tree in /usr/local/poudriere/ports/default
$ poudriere ports –c –m svn+https

# Create a jail from a snapshot
$ poudriere –j 10amd64 –v 10.0-RELEASE –a amd64
# Create a head jail from src
$ poudriere –j head-amd64 –v head –a amd64 –m svn+https

# Create a list of port origins (cat/port), 1 per line.
$ vim /usr/local/etc/poudriere.d/ports.list

# Create a public/private keypair for the repository
$ cd /etc/ssl
$ openssl genrsa -out repo.key 2048
$ chmod 0400 repo.key
$ openssl rsa -in repo.key -out repo.pub –pubout
# Configure poudriere to use your public key
$ echo "PKG_REPO_SIGNING_KEY=/etc/ssl/repo.key" >> /usr/local/etc/poudriere.conf

# Create a make.conf
$ echo "WITH_PKGNG=yes" >> /usr/local/etc/poudriere.d/make.conf

# Configure options for the build
$ poudriere options –f /usr/local/etc/poudriere.d/ports.list

# Build packages
$ poudriere bulk –j 91amd64 –f /usr/local/etc/poudriere.d/ports.list
```

signed repository and then build! (Box 2).

If you need to build multiple sets, then you should use the "-z" flag when using "options", and "bulk" commands, and also setup a `SET-make.conf` in `/usr/local/etc/poudriere.d` with any set-specific configuration.

The official Poudriere site has a guide for creating and maintaining repositories. The manual page is also online here.

There is a guide for using Jenkins to do scheduled builds of packages which is documented well in a 3 part series here.

How FreeBSD Builds Packages

The FreeBSD project used to build packages only for releases and occasionally for the STABLE branches. The old package builders used a distributed system named Portbuild. It would use a large cluster of smaller 2GB-4GB machines to build packages. This was error-prone and slow, mostly due to the older machines. A full build could still take a week. Today packages are built using single large machines using Poudriere. The FreeBSD Foundation was nice enough to purchase several 24-32 CPU 96GB machines to replace the old cluster. Using the new systems with Poudriere, the entire ports tree can be built from scratch in about 16 hours on one machine.

Packages are built for the oldest release of each branch. These packages are supposed to be ABI/KBI compatible with all future releases on those branches as well as the STABLE branch for that release. This means that packages built for 8.3 will work on 8.4 but are not guaranteed to work on 9.x. For official FreeBSD package builds, every Tuesday night a snapshot of the ports tree is made and packages begin building. Currently packages are built from 8.3, 9.1, 10.0, and head for i386 and amd64. The quarterly ports branch is also built for 10.0 on both i386 and amd64. This adds up to 10 separate package sets that must be built each week. We split these into two separate servers, one for i386 and the other for amd64. Not all 24,000 ports are built every week for every set, since Poudriere is smart enough to only build what needs to be rebuilt.

It takes just a few days for all sets to be built. After each individual package set is built, its repository is generated and signed by our signing server. An example of how we sign packages is in the pkg-repo.8 manual page. Then the packages are uploaded, using rsync, to our content delivery network for public consumption.

Meta Packages

For managing my servers I use meta-packages. These are packages that only depend on other packages and do not install any files themselves. They can be created from a port that does no actual building. By installing meta-packages onto your servers, you guarantee that each server will have the same packages installed as long as they each install only the few meta-packages you create. Using meta-packages also makes removing unneeded packages much simpler. Pkg has a feature that tracks which packages have specifically requested to be installed and which were pulled in automatically as dependencies. If you were to type `pkg install` for all packages on a system, then `pkg autoremove` would never remove anything. If instead you were to only install the one meta-package and it pulled in 100 dependencies, then in the future updated the meta-package to no longer require some other package, `pkg autoremove` would properly detect and remove that package.

I take this a step further and have role-based meta-packages. For example, I have a base meta-package that contains all packages that all my servers need. This would be things such as the configuration management tool. I then have meta-packages for each type of server that depend on only what it needs and the base meta-package. For example, my DNS server uses the dns-server meta-package that depends on bind and some other DNS tools and the base meta-package. My web application jails all have a web-server meta-package that depends on PHP, nginx and the base meta-package. This simplifies management on the servers, as only a handful of packages need to be explicitly installed and monitored. My method is to use a ports tree overlay to create my meta-packages, but you could also just create the packages directly by using the pkg manifest format. For the ports tree overlay, I just rsync my git-tracked tree over the top of an SVN checkout of ports. For Poudriere only the meta-packages need to be specified to build and it will then build all needed dependencies as well. I have a guide on my blog for managing role based servers with meta-packages.

Deployment

The official FreeBSD package URL uses an SRV DNS record to advertise which mirrors are available, but otherwise is just plain HTTP. If your network is completely internal there is not much to do for deployment. All that is needed is an internal FTP or HTTP server to serve up the package repository. The servers I host are spread out all over the world and are not in one network. At first I tried using just one HTTP server for serving the packages, but quickly found that updating all of them at once would severely slow everything down. If you have a large pipe this may still be an option. I ended up using Amazon S3 and have had a much better experience. I wrote a bulk hook for my builds using s3sync to upload the packages. I host 5GB of package sets on there and update servers weekly. This comes out to just a few dollars or less a month for updating my 20 servers.

To configure a server to use your repository, create a `/usr/local/etc/pkg/repos/MYREPO.conf` file, and also place your repository public key in `/usr/local/etc/pkg/repos/MYREPO.pem` (Box 3).

This configuration requires setup of ABI symlinks in the repository. This is a one-time operation. It allows you to use the same repository configuration on all servers without changing which release and arch it uses. Pkg will change the value of ABI when it fetches packages. It should look something like this (Box 4).

As for keeping servers up-to-date, since I am a Ports committer and also a Pkg developer, I like to observe all upgrades that I can to find any issues. I'm also just paranoid and like to make sure upgrades go smoothly, so I manually run `pkg upgrade` on my servers and don't use an automated crontab for it. My systems have a lot of applications built outside of ports, so I must save shared libraries until those applications can be rebuilt. This is similar to what portupgrade and portmaster with the "-p" flag do. An example of this can found on my github. Manually running my upgrade script is not a problem for me since I only maintain a handful of servers. However, Pkg is supported by puppet, salt and ansible. Do be warned though that some manual intervention is still required with package upgrades occasionally. This usually only occurs anymore when the origin of a package changes and requires running `pkg set -o old/origin:new/origin`. The most common case is when something like Perl is updated. You can detect this case and other cases of conflicting packages in a script by running pkg `upgrade –Fy` which will list all conflicting packages in the upgrade. The ports framework and Pkg currently do not have a means to handle replaced packages automatically, but eventually will. These cases are still currently documented in the `/usr/ports/UPDATING` file. You can keep an eye on this file in ports svnweb.

Pkg discussion takes place on the freebsd-pkg mailing list and on IRC in #pkgng on Freenode. Poudriere discussion takes place on IRC in #poudriere on Freenode. Feel free to stop by with any questions or ideas you have. •

```
MYREPO: {
  url:      "http://url.to.your.repository/${ABI}",
  enabled:  true,
  signature_type: "pubkey",
  pubkey:  "/usr/local/etc/pkg/repos/MYREPO.pem"
}

# Optionally disable the FreeBSD repo.
FreeBSD: {
  enabled:  false
}
```

Box 3. Typical pkg repository configuration

Bryan Drewery has managed shared hosting services with FreeBSD since 2004. He joined the project in 2012 as a Ports committer, is a member of Portmgr and has recently become a Src committer. He is the current upstream maintainer of Portupgrade, Portmaster, and a developer on Pkg and Poudriere. In Portmgr, he helps with the Ports framework, managing the package build systems, package building and testing ports patches on them.

```
# ls -al /usr/local/poudriere/data/packages
total 19
drwxr-xr-x   7 root  wheel  15 Jul 11  2013 ./
drwxr-x—x  26 root  wheel  32 Mar  2 01:13 ../
drwxr-xr-x   2 root  wheel   2 Jul  8  2013 10amd64/
drwxr-xr-x   7 root  wheel  39 Mar  2 10:05 83amd64/
drwxr-xr-x   7 root  wheel  39 Mar  2 10:30 83i386/
lrwxr-xr-x   1 root  wheel   7 Jul  8  2013 freebsd:10:x86:64@ -> 10amd64
lrwxr-xr-x   1 root  wheel   6 Jan 26  2013 freebsd:8:x86:32@ -> 83i386
lrwxr-xr-x   1 root  wheel   7 Jan 26  2013 freebsd:8:x86:64@ -> 83amd64
```

Box 4. ABI symlink setup

From 1.x to 11.x

THE ONGOING EVOLUTION OF THE PBI Format

Since its very early days PC-BSD has used a unique form of package management, known as PBI or

PUSH BUTTON INSTALLER

BY KRIS MOORE

While this PBI format has changed and evolved with each major release of PC-BSD, the basic concept and principles have remained the same: to provide a way for applications to be installed on a system, in a self-contained manner, without introducing messy dependency resolution issues. In principle, this means that a user can safely upgrade or downgrade an application, such as Firefox, without having to worry about changes being made to the packages which make up the rest of the system, such as X, GTK, KDE, and others.

Figure 1 provides a simplified version of what a typical dependency-driven package system can look like. This is the model most common to both FreeBSD and Linux systems. While it has some benefits, such as reduced disk space, it also introduces a large element of complexity for any updating system to cope with. When a user wants to initiate an upgrade of a package, often it requires the package management system to resolve the upgrade of a tangled web of dependencies, which could easily touch hundreds of packages in a typical desktop application. In the example of Firefox, this may leave a new user perplexed as to why bits of seemingly "unrelated" packages have to be changed, such as GTK, Gnome, and others. Assuming the upgrade of all the packages is done properly, we are still left with another potential problem, that of new bugs or regressions. While an experienced computing user may be able to find a workaround to these problems, a more casual user will be left wondering why a simple update to their web-browser means that some else in their desktop stopped working.

Figure 2 shows a simplified version of how the PBI system interacts with the other software installed on the system. In this case, each application has been distributed as a "bundle," which is installed into its own directory, such as /usr/pbi/firefox, without affecting the existing package layout on the system. This makes the process of upgrading an application as simple as updating the contents of the bundle, either as a binary differential update or a complete bundle replacement. By eliminating the reliance on dependencies, the user is now able to add, remove, and update applications at will, knowing that at worst, only that particular application will be affected.

In PC-BSD, moving applications to this model has greatly improved system reliability and consistency. However, it has presented a number of technical hurdles to overcome. The first hurdle was dealing with the system bloat that comes with having many copies of the same files installed in different PBI bundles. In PC-BSD 9, this was addressed with the implementation of a "hash" directory. This directory is created and managed by a PBI daemon, which tracks shared binaries and libraries between PBI bundles.

In Figure 3, two PBI bundles both include an identical copy of libfoo.so.1. In this case, the PBI daemon moves the library into the hash-directory, with the file's checksum attached. It then creates a hard-link of the file back into each PBI bundle.

During the upgrade of a PBI bundle, the daemon again tracks the bundle's shared binaries

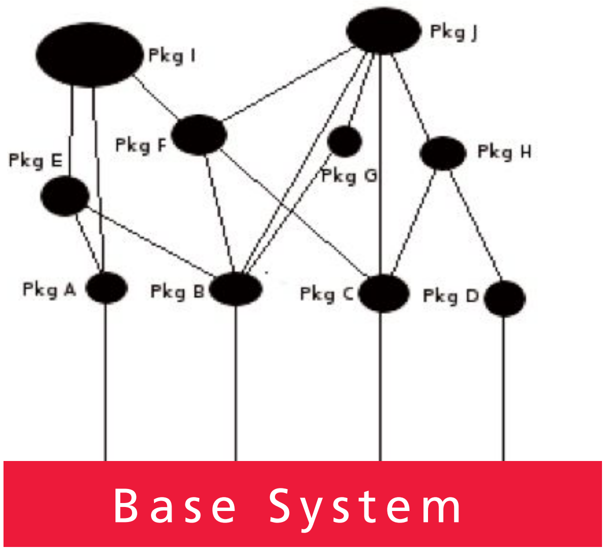


Fig. 1. A Typical Dependency-driven Package System.

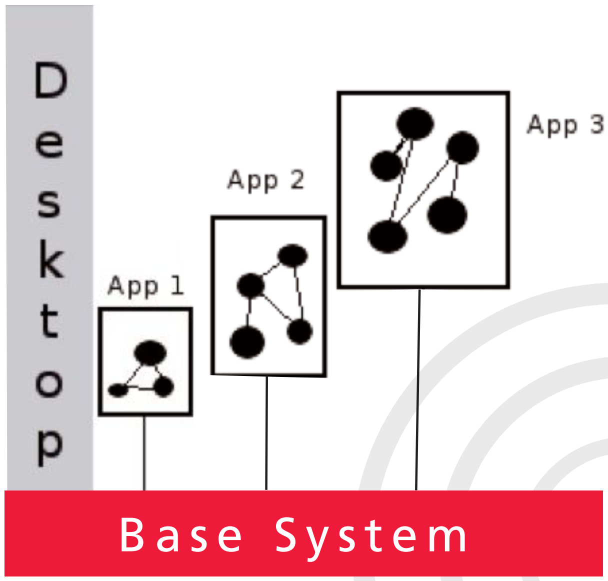


Fig. 2. PBI Application Bundles.

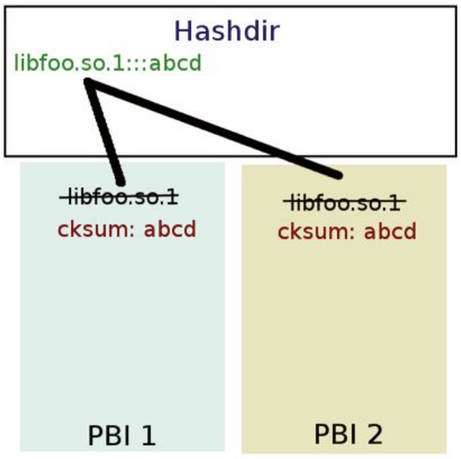


Fig. 3. Identical Files Linked to the Hash-directory.

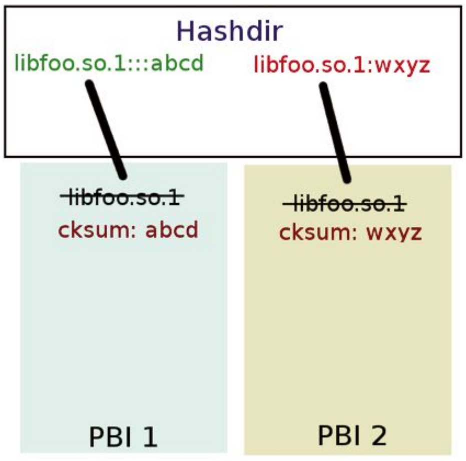


Fig. 4. Installing or Updating a Bundle with New Libraries.

and libraries. As seen in Figure 4, when a different version of an existing library is detected, the hash-directory is populated with a copy of the new library, which is again hard-linked back into the existing PBI bundle. The PBI daemon continues to monitor this directory, and when a file's hard-link counter drops to one instance, it is removed.

For the recently released PC-BSD 10.0, the PBI system has again undergone some refinement. For all previous editions of PBIs, applications were specially compiled from the ports tree using a custom LOCALBASE setting. This was done in order to force applications and libraries to behave in a self-contained manner, by having them look for their relevant data in the /usr/pbi/<appname> directory instead of the default system "/usr/local" directory. While this provided a working solution to self-containment, it presented some unique challenges. First, it exposed a number of bugs in ports and applications which expected their data to live in the system directories. Fixing these issues could range from simple to complex and proved to be very time consuming. Second, since packages were compiled with a custom LOCALBASE, each PBI needed to be compiled from scratch. This meant builders had to spend many cycles re-compiling the same libraries just to get some different location variables linked into the resulting binaries, libraries, and configuration files.

During the 9.x life-cycle, a number of enhancements were introduced to ease this building burden, such as ccache support, package caching, and more. However, going into the 10.0 release process we knew that for our PBI repositories to scale in size, we would have to fundamentally fix this problem. With the new pkgng and poudriere utilities making it easy to build the entire ports tree from scratch, we began looking at ways in which we could build PBIs from a single pkgng repository, but still keep the "self-contained" principles intact. The idea of being able to simply assemble a PBI from packages vs source building had the potential of reducing the build time for each PBI from hours down to a few minutes. However, this still left us with the run-time issues to solve. We experimented with many different options from using jails to running string replacement on binaries, but none of them proved a viable option. When looking into the idea of using jails, I thought: If jails can be used to virtualize an entire FreeBSD system, would it be possible to do some sort of "jail-lite," which only virtualizes the contents of "/usr/local," which is where our packages live? I had also seen several jail implementations that used some tricks with nullfs mounts to share a single FreeBSD world between multiple jails.

These ideas all came together when I began playing with the jailme utility in the ports tree. Instead of using calls to execute applications inside a jail, I modified the utility to do some nullfs mounting, replaced the calls to the jail with chroot, and created a "virtual" /usr/local space for a PBI to execute. And thus the idea of "PBI containers" was born. This immediately granted us the benefits that we sought: the ability to use a single set of packages, compiled with the standard /usr/local LOCABASE for assembling any number of PBI files. Additionally, it greatly reduces the complexity required for running applications, since we don't have to "force" applications to only load libraries from their own /usr/pbi directory. When the PBI is executed, it will only have access to its own files located in /usr/local.

This virtual /usr/local is accomplished is by using a wrapper binary in the front of the PBIs target executables. At launch of a target PBI, this wrapper will first check if the "virtual" container environment has been created. If not, it will proceed to perform some nullfs mounts, creating a replica of the running system in /usr/pbi/.mounts/<app>. Then the PBIs own /usr/local replacement will be mounted into this directory, replacing the systems version. With the virtual container now created, the wrapper binary will then re-create the containers ldconfig hints files in /var/run, preparing the new /usr/local directory for execution. Lastly the wrapper will chroot into the environment and execute the target application as called. This entire process can be done in usually a second or less, and only needs to be done the first time a PBI is run, making subsequent execution nearly instantaneous. The container environment stays active until the system is rebooted or the PBI is removed.

For applications that need to run commands outside the container, some callback mechanisms were added to allow running other PBIs or commands in the system's /usr/local space. These are mapped to the xdg-open and openwith commands and are typically used to open a file with user-specified application. For example, this would be used in the case of clicking a URL in some mail application and having that URL be passed along to the users default web browser.

With the runtime problems now solved, we went back and began to look at the benefits to the build infrastructure that this change bought us. The PC-BSD PBI repository for the 9.x series had grown to well over a thousand PBIs, and doing a complete rebuild from scratch easily takes 3-4 weeks on a modern server. This change reduced the build time of the same set of applications to only 48 hours!

With the release of PC-BSD 10.0 now complete, thought is already being given to the next stage of PBI evolution for version 11.0 or possibly 12.0. With the library de-duplication issue solved, and now the implementation of PBI containers, next on the roadmap is finding ways to reduce the download size of PBIs. Since PBIs are entirely self-contained, the total download size for each PBI is very large, often with the same files and libraries contained in each PBI file. While the de-duplication which occurs post-install fixes this issue for installed applications, we also want to find ways to reduce the total download size of a PBI file. This not only will save much downloading time for end-users, but also can reduce the total PBI storage footprint on our servers.

One possible solution is to make the PBI simply a "meta-file." This file would provide desktop and mime information, as well as a list of the packages that comprise the PBI. During the installation, this package list can be parsed and used as a blueprint for reassembling the PBI into a container, using cached pkgng packages. Since many of the packages used by PBIs are identical, this allows us to only fetch the missing packages needed for a specific PBI to be installed. Once the packages are all cached on disk, the installer can then assemble them back into an installed PBI. This method would greatly shrink the data being sent over the wire, but also opens up interesting ideas such as making PBIs customizable, by adjusting the particular packages that are reassembled into the container. This can all be done while preserving the integrity of the main system packages, which comprise a desktop or server installation, and other PBI containers. For users who still want the "offline" experience of copying a single PBI file to USB and installing it on a non-connected system, we can provide mechanisms to assemble the various meta-data and packages into a single file for installation.

While the PBI format has evolved greatly over the years, we are confident that it will continue to improve with each major release. All the while staying true to the vision of making the installation and upgrade of applications as simple and risk-free as possible. Users who wish to participate in this discussion and development are encouraged to get in touch with us on the PC-BSD developers mailing list. ●

Kris Moore is the founder and lead developer of the PC-BSD project. He is also the co-host of the popular BSDNow video podcast. When not at home programming, he travels around the world giving talks and tutorials on various BSD-related topics at Linux and BSD conferences alike. He currently lives in Tennesee (USA) with his wife and five children and enjoys playing bass guitar and video gaming in his (very limited) spare time.

Further Reading on the PBI Format:

http://wiki.pcbsd.org/index.php/PBI_Manager/10.0

http://wiki.pcbsd.org/index.php/PBI9_Format

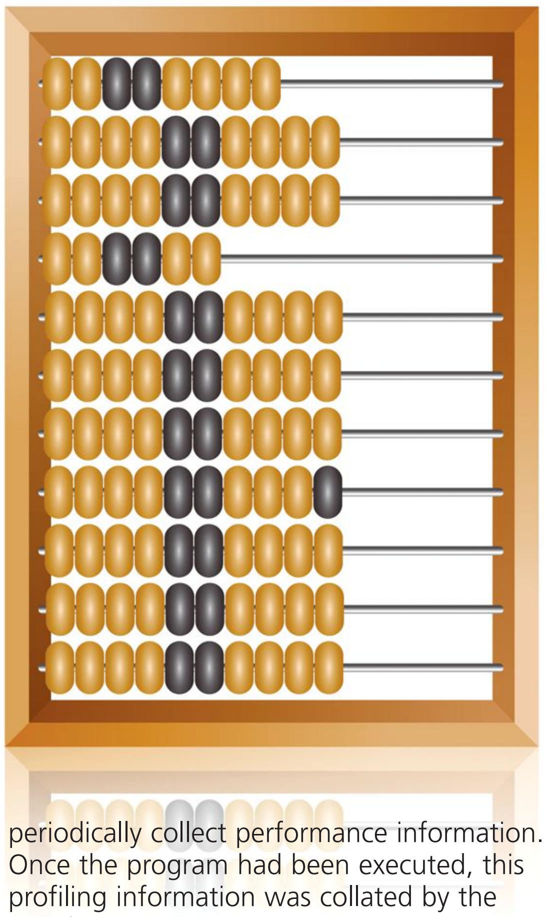
http://www.bsdcan.org/2008/schedule/events/81.en.html

http://bsdtalk.blogspot.com/2008/02/bsdtalk141-pbi4-with-kris-moore.html

http://2011.eurobsdcon.org/talks.html#moore

UNDERSTANDING APPLICATION AND SYSTEM PERFORMANCE WITH HWPMC(4)

BY GEORGE NEVILLE-NEIL



Architecture and memory have become more complex over the last several years. This added complexity has made it harder than ever to understand software performance.

Fortunately, CPU designers have added features to their chips to enable developers and administrators to better understand the performance of software with a very small amount of overhead. The Hardware Performance Monitoring Counters, hwpmc(4), driver and associated tools on FreeBSD provide software developers—and anyone else who is interested in system performance—a way to better understand how efficiently their software is utilizing the underlying hardware, their applications and the operating system itself. The hwpmc subsystem takes advantage of hardware specific registers set aside by the CPU designer purely for the purpose of understanding the run time performance of the system.

It Was 20 Years Ago...

In early versions of BSD software, performance was measured using a software based profiling system and an associated tool, named gprof. The g stands for graph, as in call graph. Software that was to be profiled was compiled with a set of flags indicating that the resulting program should have special hooks to tell the operating system to periodically collect performance information. Once the program had been executed, this profiling information was collated by the gprof program and presented to the user.

A software based profiling system has several problems. First, software profiling is computationally expensive. Depending on how often the profiler runs, it may introduce an overhead penalty on the order of 10% to 20%. For a short program, the profiling can easily outweigh the code being profiled, resulting in measurements that are useless. A second problem is that a software based profiling solution changes the flow of the resulting binary program, meaning that the code being profiled is not a one to one representation of the final software as it would be shipped to a customer. While it is possible to ship profiled binaries to customers, the overhead incurred in a profiled binary would result in worse overall system performance, which would be unacceptable. The third problem presented by software based profiling is that it is impossible for an end user to measure the performance of their system on their own. A customer with a non-profiled binary application to run has no way of adding profiling to the binary to find out if the program is a source of system inefficiency. Hardware based performance solutions ameliorate some of these problems.

Hardware Based Performance Monitoring Counters

As CPUs got to be more densely packed with transistors, and as the feature sets of CPUs got larger, it became possible for CPU designers to include specific registers to count events relating to system performance. At first the types and numbers of events that could be counted were small, with only a handful of events and one or two counting registers. It was only possible to count instructions that were executed or the number of level one cache misses. On a modern Intel CPU, hundreds of event types can be counted, and there are enough counting registers to record 7 different events simultaneously.

When working with hardware based performance monitoring counters there are a few pieces of terminology to keep in mind. An "event" is anything that the chip can count for you, such as the number of instructions retired, branch predictions that were missed, cycles required to fetch memory, etc. A counting register is a place where an event can be counted. Events can be recorded in two different modes, and counted in two different scopes. An event may be simply "counted" or the CPU may be configured to interrupt the operating system when a counter has hit a set level, an event which the hwpmc(4) driver records in its log for later analysis. A counted event gives a raw number that tells the programmer how many of some event happened over a particular unit of time. A sampled event is more complicated than a counted event. In event sampling, the system is programmed to take a sample of the instruction pointer and possibly the program's call chain, whenever some number of events has occurred. Sampling allows the system to show where an event occurred in the software, helping the programmer pinpoint the source of a performance problem. Events can be counted and sampled in one of two scopes. Process scope records events only when the target program is currently executing. System scope records events at all times and when coupled with sampling mode will show the performance not only of the program that is being tested, but of all programs in the system, including the operating system itself. It is possible to count events in either system or process mode as well as sample events in system or process mode.

Measuring Performance with hwpmc

The easiest way to learn to use hwpmc in your own programs is by trying with a few contrived examples. The unixbench[cite] system of benchmarks is a well-known, easy to understand, set of programs that try to determine the speed of both software and hardware. We will measure the performance of several programs from unixbench with the pmc tools in order to give some clear examples.

Before working with the hwpmc(4) driver it must be loaded into your kernel. To measure system level performance you will also need root privileges on the machine on which you wish to use hwpmc. The default (GENERIC) kernel does not have hwpmc loaded at boot time. In order to load hwpmc issue the following commands as root (Figure1).

Fig. 1

```
# kldload hwpmc

hwpmc: SOFT/16/64/0x67<INT,USR,SYS,REA,WRI> TSC/1/64/0x20<REA>
IAP/4/48/0x3ff<INT,USR,SYS,EDG,THR,REA,WRI,INV,QUA,PRC>
IAF/3/48/0x67<INT,USR,SYS,REA,WRI>
```

When the hwpmc(4) driver is loaded it reports the number, type and width of the counting registers it finds on the CPU. The output varies widely from processor to processor, even within the same vendor family. In the example shown above there are sixteen soft registers, one time stamp counter, four programmed counters and three fixed counters. Certain events can only be counted in certain types of registers, and you will be given an error if you try to count events in a register that does not accept the event you are asking for. For the most part you only need to know the number of registers in each class, as the system attempts to assign events correctly when you ask for them. If a user tries to count four events that are only possible in the fixed type registers (IAF) then the tools will report an error and exit without counting any events.

The types of events that can be counted are listed with the `pmccontrol -L` command. There are 194 possible events that can be counted on this host, but don't worry, we will not go through all of them.

One of the first program measurements that is typically made is the raw number of instructions that it executes in a particular amount of time. The event, INSTR_RETIRED_ANY, counts instructions executed. The term retired is used because the end of executing an instruction is called retiring in CPU parlance.

In order to count or sample events the

```
int main(argc, argv)
int   argc;
char  *argv[];
{
 ... (Intentionally Left Blank) ...

      while(1) {
            mov(disk,1,3);
            iter++;
            }

      exit(0);
}

void mov(int n, int f, int t)
{
      int o;
      if(n == 1) {
            num[f]–;
            num[t]++;
            return;
      }
      o = other(f,t);
      mov(n-1,f,o);
      mov(1,f,t);
      mov(n-1,o,t);
}
```

`pmcstat` command is used.

Figure 2 shows a simple, cumulative, count of instruction's retired when the **hanoi** benchmark is run for ten seconds. We need not concern ourselves with the details of the **hanoi** program at this point, but we will dig into it more later. The first column is the number of instructions that were retired during the entire run of the program, the last number showing the cumulative total of 27527655088. In the ten seconds that the **hanoi** program was running it executed 1343590 loops, and this output is from the **hanoi** program itself, it has nothing to do with hwpmc.

Examples in this article were run on a Lenovo Thinkpad X230 with a Core i7 running at 2.9GHz.

For comparison, we ran the **hanoi** program without the performance monitoring system. Based on the number of loops that **hanoi** was able to execute in ten seconds (Figure 3) we see that hwpmc introduces less than 2% of overhead.

One common measure of code efficiency is Cycles Per Instruction (CPI) which is derived by counting both instructions retired and the number of cycles during a test run, and then dividing the two results.

The command in Figure 4 counts instructions retired and clock cycles simultaneously. Dividing the clock cycles by the number of instructions retired gets us a CPI of 0.44, which, according to Intel's optimization manuals, is an acceptable value. Higher values of CPI, for instance those greater than 1, indicate that instructions are taking longer than they ought to. For a full discussion of CPI and general performance tuning using Intel's PMC events see:

(http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf)

Counting events gives an overall idea of the efficiency of a complete program. To look more deeply into where a piece of code is spending its time we need to use sampling mode.

The command shown in Figure 5 uses the instruction retired event from the previous example, but switches to sampling mode, indicated by the capital P command line switch, and stores the resulting output in a log file in /tmp/hanoi.log. Without the output option, the entire log would be dumped to stdout, which wouldn't be very useful.

Once the event collection is complete we analyze the log file with pmcstat to see what functions were taking up the most instructions.

We process the collected log file in Figure 6 to produce a graph file. The output in the graph file (hanoi.graph) shows the functions that took up the largest percentage of events, starting with the largest and proceeding to the smallest.

The output in Figure 7 shows that the mov() routine, see the code in Listing X, takes up the largest number of samples, whereas the main() routine for the program had very few. The result is what we'd expect from this program. Output from pmcstat can be shown in another way, as gprof(1) output `pmcstat -R /tmp/hanoi.log -g`. (Figure 7).

Processing the same log with the -g argument creates a per-event directory, INSTR_RETIRED_ANY/ which contains output files for each program, library, and the kernel that were in use when the samples were taken.

Processing the **hanoi**.gmon file gives the output shown in Figure 8. Time, in this case, is misleading. The numbers in the seconds columns represent events counted, and not seconds, but the output is convenient and brief to read. We still see that the mov() routine is the biggest consumer of events, taking up 99.8%

```
pmcstat -C -p INSTR_RETIRED_ANY ./hanoi 10
# p/INSTR_RETIRED_ANY
          14201054051 1343590 loops
          27527655088
```

Fig. 2

```
./hanoi 10
1357889 loops
```

Fig. 3

```
pmcstat -C -p INSTR_RETIRED_ANY -p CPU_CLK_UNHALTED_CORE ./hanoi 10
# p/INSTR_RETIRED_ANY p/CPU_CLK_UNHALTED_CORE
          13318387828             5962151280 1324620 loops

          27139612697            11987228985
```

Fig. 4

```
> pmcstat -O /tmp/hanoi.log -P INSTR_RETIRED_ANY ./hanoi 10
1013645 loops
```

Fig. 5

```
> pmcstat -R /tmp/hanoi.log -G /tmp/hanoi.graph
```

Fig. 6

```
@ INSTR_RETIRED_ANY [365189 samples]

99.17%  [362173]   mov @
/usr/home/gnn/svn/headports/benchmarks/unixbench/work/unixbench-4.1.0/pgms/hanoi
 99.61%  [360744]    mov
  97.57%  [351963]     mov
   90.90%  [319928]      mov
   09.10%  [32035]       main
  02.43%  [8781]       main
   100.0%  [8781]        _start
 00.39%  [1429]      main
  100.0%  [1429]       _start
```

Fig. 7

```
> ls
hanoi.gmon                 kernel.gmon                libc.so.7.gmon
hwpmc.ko.gmon              ld-elf.so.1.gmon
> gprof ../hanoi hanoi.gmon
granularity: each sample hit covers 4.00673 byte(s) for 0.00% of 362181.00 seconds

 %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.8  361564.31 361564.31    42245  8558.75  8558.75  mov [3]
  0.0  361572.29      7.99    10210     0.78 35413.54  main [1]
  0.0  361572.29      0.00        0    0.00%            _start [2]
```

Fig. 8

of all events found relating to the program.

Up to this point we have shown the hwpmc system working only in process scope. The **hanoi** program is meant to show only the performance of the CPU and has no interaction with the underlying operating system. We will now move on to the syscall benchmark which shows the performance of one aspect of the operating system itself, the speed of a system call.

We see the main loop of the syscall program in Listing Y. The benchmark measures the speed of the operating system's system calls by repeatedly duplicating a file descriptor, getting the process ID and the user ID, and setting the umask. Each of these calls does a very small amount of work compared to the work required to enter and exit the kernel, and therefore

Y

```
    while     (1) {
               close(dup(0));
               getpid();
               getuid();
               umask(022);
               iter++;
         }
```

make good targets for measuring the overhead of the system call mechanism itself.

We collect the samples and generate a graph file in Figure 9. The graph file contains over 5000 lines of output, including functions that have no relation to the syscall benchmark program. For this example, we are using system-wide scope for even collection, and so we have collected events for all the various processes currently executing on the system, including Emacs, in which this article is being written. The first several lines of the graph file are shown in Figure 10.

The largest number of samples, 10%, come from the witness_unlock() kernel routine. As we move down the graph we see the constituent components that contributed to the 12263 events recorded against witness_unlock(), including do_dup(), closefp(), and sys_umask() which are the kernel side routines that are called by the dup(), close() and umask() system calls. The cheapest system calls, getuid() and getpid() do not occur until much farther down in the file. An interesting comparison is the number of events that are counted against libc vs. those that are counted against the kernel.

Figure 11 shows the number of events counted against the library version of getuid vs. those counted against the kernel side of the system call, sys_getuid. The boilerplate in the C library is minimal compared to that seen in the kernel, which tells us that the largest speed up in this code would be afforded by improving the kernel side code. Both getpid() and getuid() are trivial, but are used in benchmarks to determine the overhead of a system call.

Counters Counters Everywhere

Originally only available on a small number of Intel and AMD processors, hwpmc has now been extended to cover ARM and MIPS processors as well, giving developers the ability to profile their code on popular embedded systems. The events and counters are architecture specific, but the basic concepts remain the same. The hwpmc system also provides convenient aliases for common events, such as cycles, for whatever the CPU's cycle counter is, and instructions, for instructions retired. Aliases are always lower case, and architecture specific counters, like INSTR_RETIRED_ANY, are upper case. Event aliases are present across almost all processors supported by FreeBSD, which makes writing portable performance analysis scripts easier.

As new processors are put into the market by CPU vendors the hwpmc system gets extended to add support for newer events and newer sets of counting registers. If you're trying to understand the performance characteristics of the software you're writing or the system as a whole, the hwpmc system and its tools are a great place to start. •

George Neville-Neil works on networking and operating system code for fun and profit. He also teaches various courses on subjects related to computer programming. His professional areas of interest include code spelunking, operating systems, networking, and security. He is the co-author with Marshall Kirk McKusick of *The Design and Implementaion of the FreeBSD Operating System* and is the columnist behind ACM Queue magazine's *Kode Vicious*. Neville-Neil earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts. He is a member of the ACM, the Usenix Association, and the IEEE. He is an avid bicyclist and traveler and currently resides in New York City.

```
> sudo pmcstat -O /tmp/syscall.log -S INSTR_RETIRED_ANY ./syscall 10
3232709 loops
> sudo pmcstat -R /tmp/syscall.log -G /tmp/syscall.graph
CONVERSION STATISTICS:
 #exec/elf                                   1
 #samples/total                               262735
 #samples/unclaimed                          6
 #samples/unknown-function                  41
 #callchain/dubious-frames                  6
```

Fig. 9

```
@ INSTR_RETIRED_ANY [113032 samples]

10.85%  [12263]    witness_unlock @ /boot/kernel/kernel
 54.50%  [6683]      _sx_xunlock
  47.36%  [3165]       do_dup
   100.0%  [3165]        amd64_syscall
  32.71%  [2186]       closefp
   100.0%  [2186]        amd64_syscall
  19.93%  [1332]       sys_umask
   100.0%  [1332]        amd64_syscall
```

Fig. 10

```
01.45%  [1642]     sys_getuid @ /boot/kernel/kernel
00.47%  [526]      getuid @ /lib/libc.so.7
```

Fig. 11

ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.

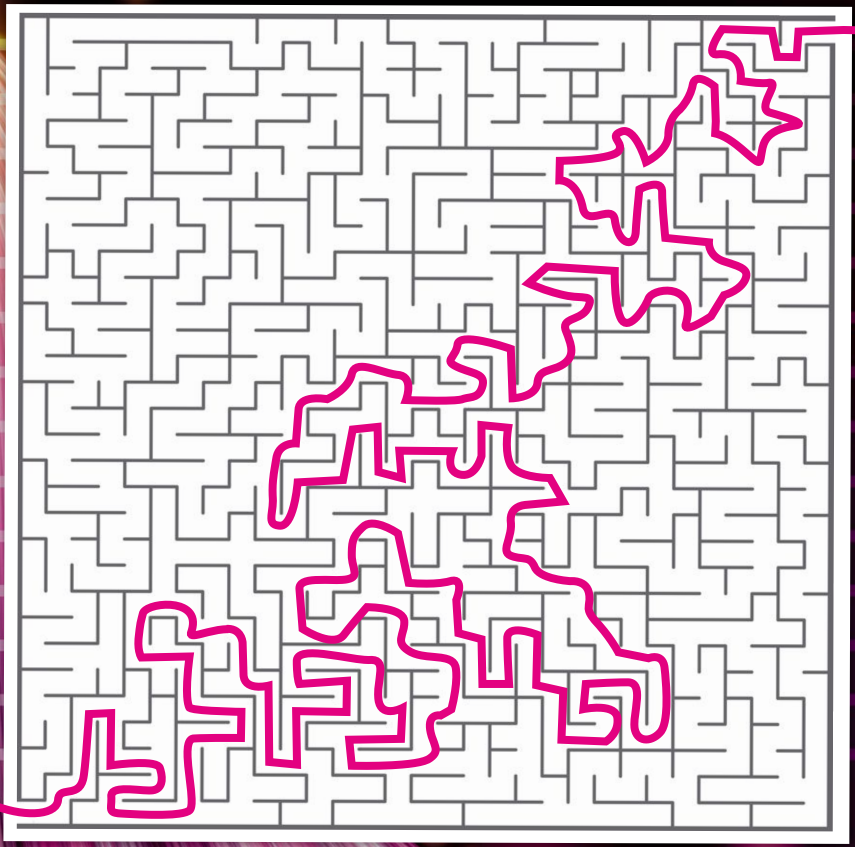
We're Hiring!

With offices around the world, we likely have a job for you! Please visit our website at http://www.emc.com/careers or send direct inquiries to annie.romas@emc.com.

EMC² ISILON

JOURNALED soft-updates

The soft-updates **DEPENDENCY TRACKING SYSTEM** was adopted by FreeBSD in 1998 as an alternative to the popular journaled file system technique.



By
Marshall Kirk McKusick
and Jeff Roberson

While the runtime performance and consistency guarantees of soft updates are comparable to journaled filesystems [Seltzer, Ganger, McKusick et al, 2000], a journaled filesystem relies on an expensive and time-consuming background filesystem recovery operation after a crash.

This article outlines a method for eliminating the expensive background or foreground whole-filesystem check operation by using a small journal that logs the only two inconsistencies possible in soft updates. The first is allocated but unreferenced blocks; the second is incorrectly high link counts. Incorrectly high link counts include unreferenced inodes that were being deleted and files that were unlinked but open [Ganger, McKusick, & Patt, 2000]. This journal allows a journal-analysis program to complete recovery in just a few seconds independent of filesystem size.

After a crash, a variant of the venerable fsck program runs through the journal to identify and free the lost resources. Only if an inconsistency between the log and filesystem is detected is it necessary to run the whole-filesystem fsck. The journal is tiny, 16 Mbytes is usually enough, independent of filesystem size. Although journal processing needs to be done before restarting, the processing time is typically just a few seconds and in the worst case a minute.

Compatibility with Other Implementations

It is not necessary to build a new filesystem to use soft-updates journaling. Journaling is enabled via tunefs and only requires a few spare superblock fields and 16 Mbytes of free blocks for the journal. These minimal requirements make it easily enabled on existing FreeBSD filesystems. The journal's filesystem blocks are placed in an inode named .sujournal in the root of the filesystem and filesystem flags are set such that older non-journaling kernels will trigger a full filesystem check when mounting a previously journaled volume. When mounting a journaled filesystem, older kernels clear a flag that shows that journaling is being done, so that when the filesystem is next encountered by a kernel that does journaling, it will know that the journal is invalid and will ensure that the filesystem is consistent and clear the journal before resuming use of the filesystem.

Journal Format

The journal is kept as a circular log of segments containing records that describe metadata operations. If the journal fills, the filesystem must complete enough operations to expire journal entries before allowing new operations. In practice, the journal almost never fills.

Each journal segment contains a unique sequence number and a timestamp that identifies the filesystem mount instance so old segments can be discarded during journal processing. Journal entries are aggregated into segments to minimize the number of writes to the journal. Each segment contains the last valid sequence number at the time it was written to allow fsck to recover the head and tail by scanning the entire journal. Segments are variably sized as some multiple of the disk block size and are written atomically to avoid read/modify/write cycles in running filesystems.

The journal-analysis has been incorporated into the fsck program. This incorporation into the existing fsck program has several benefits. The existing startup scripts already call fsck to see if it needs to be run in foreground or background. For filesystems running with journaled soft updates, fsck can request to run in foreground and do the needed journaled operations before the filesystem is brought online. If the journal fails for some reason, it can instead report that a full fsck needs to be run as the traditional fallback. Thus, this new functionality can be introduced without any change to the way that system administrators start up their systems. Finally, the invoking of fsck means that after the journal has been processed, it is possible for debugging purposes to fall through and run a complete check of the filesystem to ensure that the journal is working properly.

The journal entry size is 32 bytes, providing a dense representation allowing for 128 entries per 4-Kbyte sector. The journal is created in a single area of the filesystem in as contiguous an allocation as is available. We considered spreading it out across cylinder groups to optimize locality for writes, but it ended up being so small that this approach was not practical and would make scanning the entire journal during cleanup too slow.

The journal blocks are claimed by a named immutable inode. This approach allows user-level access to the journal for debugging and statistics gathering purposes as well as providing backwards compatibility with older kernels that do not support journaling. We have found that a journal size of 16 Mbytes is enough in even the most tortuous and worst-case benchmarks. A 16-Mbyte journal can cover over 500,000 namespace operations or 16 Gbyte of outstanding allocations (assuming a standard 32-Kbyte block size).

Modifications that Require Journaling

This subsection describes the operations that must be journaled so that the information needed to clean up the filesystem is available to fsck.

Increased Link Count

A link count may be increased through a hard link or file creation. The link count is temporarily increased during a rename. Here, the operation is the same. The inode number, parent inode number, directory offset, and initial link count are all recorded in the journal. Soft updates guarantees that the inode link count will be increased and stable on disk before any directory write. The journal write must occur before the inode write that updates the link count and before the bitmap write that allocates the inode if it is newly allocated.

Decreased Link Count

The inode link count is decreased through unlink or rename. The inode number, parent inode, directory offset, and initial link count are all recorded in the journal. The deleted directory entry is guaranteed to be written before the link is adjusted down. As with increasing the link count, the journal write must happen before all other writes.

Unlink While Referenced

Unlinked yet referenced files pose a problem for journaled filesystems. In POSIX, an inode's storage is not reclaimed until after the final name is removed and the last reference is closed. Simply leaving the journal entry valid while waiting for applications to close their dangling references is untenable as it will easily exhaust journal space. A solution that scales to the total number of inodes in the filesystem is required. At least two approaches are possible, a replication of the inode allocation bitmap, or a linked list of inodes to be freed. We have chosen to use the linked-list approach.

In the linked-list case, which is employed by several filesystems (xfs, ext4, etc.), the superblock contains the inode number that serves as the head of a singly linked list of inodes to be freed, with each inode storing a pointer to the next inode in the list. The advantage of this approach is that at recovery time fsck need only examine a single pointer in the superblock that will already be in memory. The disadvantage is that the kernel must keep an in-memory, doubly-linked list so that it can rapidly remove an inode once it is unreferenced. This approach ingrains a filesystem-wide lock in the design and incurs non-local writes when maintaining the list. In practice we have found that unreferenced inodes occur rarely enough that this approach is not a bottleneck.

Removal from the list may be done lazily but must be completed before any re-use of the inode. Additions to the list must be stable before reclaiming journal space for the final unlink, but otherwise may be delayed long enough to avoid needing the write at all if the file is quickly closed. Addition and removal involve only a single write to update the preceding pointer to the following inode.

Change of Directory Offset

Any time a directory compaction moves an entry, a journal entry must be created describing the old and new locations of the entry. The kernel does not know at the time of the move whether a remove will follow it, so currently all offset changes are journaled. Without this information fsck would be unable to disambiguate multiple revisions of the same directory block.

Block Allocation and Free

When performing either block allocation or free, whether it is a fragment, indirect block, directory block, direct block, or extended attributes the record is the same. The inode number of the file and the offset of the block within the file are recorded using negative offsets for indirect and extended attribute blocks. Additionally, the disk block address and number of fragments are included in the journal record. The journal entry must be written to disk before any allocation or free.

When freeing an indirect block only the root of the indirect block tree is logged. Thus, for truncation we need a maximum of 15 journal entries, 12 for direct blocks and 3 for indirect blocks. These 15 journal entries allow us to free a large amount of space with a minimum of journaling overhead. During recovery, fsck will follow indirect blocks and free any descendants including other indirect blocks. For this algorithm to work, the contents of the indirect block must remain valid until the journal record is free so that user data is not confused with indirect block pointers.

Additional Requirements of Journaling

Some operations that had not previously required tracking under soft updates need to be tracked when journaling is introduced. This subsection describes these new requirements.

Cylinder Group Rollbacks

Soft updates previously did not require any rollbacks of cylinder groups as they were always the first or last write in a group of changes. When a block or inode has been allocated, but its journal record has not yet been written to disk, it is not safe to write the updated bitmaps and associated allocation information. The routines that write blocks with "bmsafemap" dependencies now rollback any allocations with unwritten journal operations.

Inode Rollbacks

The inode link count must be rolled back to the link count as it existed before any unwritten journal entries. Allowing it to grow beyond this count would not cause filesystem corruption, but it would prohibit the journal recovery from adjusting the link count properly. Soft updates already prevents the link count from decreasing before the directory entry is removed, as a premature decrement could cause filesystem corruption.

When an unlinked file has been closed, its inode cannot be returned to the inode freelist until its zeroed-out block pointers have been written to disk so that its blocks can be freed and it has been removed from the on-disk list of unlinked files. The unlinked-file inode is not completely removed from the list of unlinked files until the next pointer of the inode that precedes it in the list has been updated on disk to point to the inode that follows it on the list. If the unlinked-file inode is the first inode on the list of unlinked files, then it is not completely removed from the list of unlinked files until the head-of-unlinked-files pointer in the superblock has been updated on disk to point to the inode that follows it on the list.

Reclaiming Journal Space

To reclaim journal space from previously written records, the kernel must know that the operation the journal record describes is stable on disk. This requirement means that when a new file is created, the journal record cannot be freed until writes are completed for a cylinder group bitmap, an inode, a directory block, a directory inode, and possibly some number of indirect blocks. When a new block is allocated, the journal record cannot be freed until writes are completed for the new block pointer in the inode or indirect block, the cylinder group bitmap, and the block itself. Blocks pointers within indirect blocks are not stable until all parent indirect blocks are fully reachable on disk via the inode indirect block pointers. To simplify fulfillment of these requirements, the dependencies that describe these operations carry pointers to the oldest segment structure in the journal containing journal entries that describe outstanding operations.

Some operations may be described by multiple entries. For example, when making a new directory, its addition creates three new names. Each of these names is associated with a reference count on the inode to which the name refers. When one of these dependencies is satisfied, it may pass its journal entry reference to another dependency if another operation on which the journal entry depends is not yet complete. If the operation is complete, the final reference on the journal record is released. When all references to journal records in a journal segment are released, its space is reclaimed and the oldest valid segment sequence number is adjusted. We can only release the oldest free journal segment, since the journal is treated as a circular queue.

Handling a Full Journal

If the journal ever becomes full, we must prevent any new journal entries from being created until more space becomes available from the retirement of the oldest valid entries. An effective way to stop the creation of new journal records is to suspend the filesystem using the mechanism in place for taking snapshots. Once suspended, existing operations on the filesystem are permitted to complete, but new operations that wish to modify the filesystem are put to sleep until the suspension is lifted.

We do a check for journal space

before each operation that will change a link count or allocate a block. If we find that the journal is approaching a full condition, we suspend the filesystem and expedite the progress on the soft-updates work-list processing to speed the rate at which journal entries get retired. As the operation that did the check has already started, it is permitted to finish, but future operations are blocked. Thus, operations must be suspended while there is still enough journal space to complete operations already in progress. When enough journal entries have been freed, the file system suspension is lifted and normal operations resume.

In practice, we had to create a minimal sized journal (4 Mbyte) and run scripts designed to create huge numbers of link-count changes, block allocations, and block frees to trigger the journal-full condition. Even under these tests, the filesystem suspensions were infrequent and brief lasting under a second.

The Recovery Process

This subsection describes the use of the journal by fsck to clean up the filesystem after a crash.

Scanning the Journal

To do recovery, the fsck program must first scan the journal from start to end to discover the oldest valid sequence number. We contemplated keeping journal head and tail pointers, however, that would require extra writes to the superblock area. Because the journal is small, the extra time spent scanning it to identify the head and tail of the valid journal seemed a reasonable tradeoff to reduce the run-time cost of maintaining the journal head and tail pointers. So, the fsck program must discover the first segment containing a still valid sequence number and work from there. Journal records are then resolved in order. Journal records are marked with a timestamp that must match the filesystem mount time as well as a CRC to protect the validity of the contents.

Adjusting Link Counts

For each journal record recording a link increase, fsck needs to examine the directory at the offset provided and see whether the directory entry for the recorded inode number exists on disk. If it does not exist, but the inode link count was increased, then the recorded link count needs to be decremented.

For each journal record recording a link decrease, fsck needs to examine the directory at the offset provided and see whether the directory entry for the recorded inode number exists on disk. If it has been deleted on disk, but the inode link count has not been decremented, then the recorded link count needs to be decremented.

Compaction of directory offsets for entries that are being tracked complicates the link adjustment scheme presented above. Since directory blocks are not written synchronously, fsck must look up each directory entry in all its possible locations.

When an inode is added and removed from a directory multiple times, fsck is not able to correctly assess the link count given the algorithm presented above. The chosen solution is to pre-process the journal and link together all entries related to the same inode. In this way, all operations not known to be committed to the disk can be examined concurrently to determine how many links should exist relative to the known stable count that existed before the first journal entry. Duplicate records that occur when an inode is added and deleted at the same offset many times are discarded, resulting in a coherent count.

Updating the Allocated Inode Map

Once the link counts have been adjusted, fsck must free any inodes whose link count has fallen to zero. In addition, fsck must free any inodes that were unlinked, but still in use at the time that the system crashed. The head of the list of unreferenced inode is in the superblock as described earlier in this article. The fsck program must traverse this list of unlinked inodes and free them.

The first step in freeing an inode is to add all its blocks to the list of blocks that need to be freed. Next, the inode needs to be zeroed to show that it is not in use. Finally, the inode bitmap in its cylinder group must be updated to reflect that the inode is available and all the appropriate filesystem statistics updated to reflect the inodes availability.

Updating the Allocated Block Map

Once the journal has been scanned, it provides a list of blocks that were intended to be freed. The journal entry lists the inode from which the block was to be freed. For recovery, fsck processes each free record by checking to see if the block is still claimed by its associated inode. If it finds that the block is no longer claimed, it is freed.

FOSSETCON

FREE AND OPEN SOURCE EXPO AND TECHNOLOGY CONFERENCE

FOSSETCON 2014

Come out and participate in the First Annual Fossetcon 2014 Florida's Only Free and Open Source Conference. With in 10 minutes of Disney Land, Universal Studios and Epcot Center.

DAY 0 FOOD, TRAINING, WORKSHOPS AND CERTIFICATIONS

DAY 1 FOOD, KEYNOTES, EXPO HALL, SPEAKER TRACKS

DAY 2 FOOD, KEYNOTES, EXPO HALL, SPEAKER TRACKS

BSD *Friendly*

FREE FOOD, TRAINING, CERTIFICATIONS AND GIVEAWAYS!!!

SEP 11 - SEPT 13

ROSEN PLAZA HOTEL ORLANDO, FL

More info at **www.fossetcon.org**

Fossetcon 2014: The Gateway To The Open Source Community

For each block that is freed either by the deallocation of an inode, or through the identification process described above, the block bitmap in its cylinder group must be updated to reflect that it is available and all the appropriate filesystem statistics updated to reflect its availability. When a fragment is freed, the fragment availability statistics must also be updated.

Performance

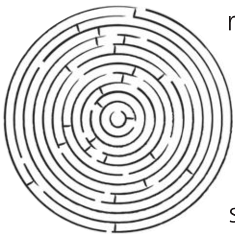
Journaling adds extra running time and memory allocations to the traditional soft-updates requirements and also additional I/O operations to write the journal. The overhead of the extra running time and memory allocations was immeasurable in the benchmarks that we ran. The extra I/O was mostly evident in the increased delay for individual operations to complete. Operation completion time is usually only evident to an application when it does an "fsync" system call that causes it to wait for the file to reach the disk. Otherwise, the extra I/O to the journal only becomes evident in benchmarks that are limited by the filesystem's I/O bandwidth before journaling is enabled.

In summary, a system running with journaled soft updates will never run faster than one running soft updates without journaling. So, systems with small filesystems—such as an embedded system—will usually want to run soft updates without journaling and take the time to run fsck after system crashes.

The primary purpose of the journaling project was to eliminate long filesystem check times. A 40-Tbyte volume may take an entire day and a considerable amount of memory to check.

We have run several scenarios to understand and validate the recovery time.

A typical operation for developers is to run a parallel buildworld. Crash recovery from this case demonstrates time to recover from moderate write workload. A 250-Gbyte disk was filled to 80% with copies of the FreeBSD source tree. One copy was selected at random and an 8-way buildworld proceeded for 10 minutes before the box was reset. Recovery from the journal took 0.9 seconds. An additional run with traditional fsck was used to verify the safe recovery of the filesystem. The fsck took about 27 minutes, or 1,800 times as long.



A testing volunteer with a 92% full 11-Tbyte volume spanning 14 drives on a 3ware RAID controller generated hundreds of megabytes of dirty data by writing random length files in parallel before resetting the machine. The resulting recovery operation took less than one minute to complete. A normal fsck run takes about 10 hours on this filesystem. ●

Marshall Kirk McKusick writes books and articles, consults, and teaches classes on Unix- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast file system and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG), overseeing the development and release of 4.3BSD and 4.4BSD. He has twice been president of the board of the Usenix Association, is currently a member of the FreeBSD Foundation Board of Directors, a member of the editorial board of ***ACM Queue*** magazine and ***The FreeBSD Journal***, a senior member of the IEEE, and a member of the Usenix Association, ACM, and AAAS. You can contact him via email at mckusick@mckusick.com.

Jeff Roberson is a consultant who lives on the island of Maui in the Hawai'ian island chain. When he is not cycling, hiking, or otherwise enjoying the island, he gets paid to improve FreeBSD. He is particularly interested in problems facing server installations and has worked on areas as varied as the kernel memory allocator, thread scheduler, filesystems interfaces, and network packet storage, among others. You can contact him via email at jroberson@jroberson.net.

REFERENCES

G. Ganger, M. McKusick, & Y. Patt, "Soft Updates: A Solution to the Metadata Update Problem in Filesystems," *ACM Transactions on Computer Systems* **18**(2), p. 127–153 (May 2000).

M. Seltzer, G. Ganger, M. K. McKusick, K. Smith, C. Soules, & C. Stein, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," *Proceedings of the San Diego Usenix Conference*, pp. 71-84 (June 2000).

Testers, Systems Administrators, Authors, Advocates, *and of course* **Programmers** *to join any of our diverse teams.*

COME JOIN THE PROJECT THAT MAKES THE INTERNET GO!

★ DOWNLOAD OUR SOFTWARE ★

http://www.freebsd.org/where.html

★ JOIN OUR MAILING LISTS ★

http://www.freebsd.org/community/mailinglists.html?

★ ATTEND A CONFERENCE ★

• *http://www.bsdcan.org/2014/* • *http://2014.eurobsdcon.org/*

The FreeBSD Project

The FreeBSD FOUNDATION

svnUPDATE

by Glen Barber

FreeBSD 10.0 is finally released, and the FreeBSD developers have already begun working on the next round of excellent features that are sure to appear in future releases.

CASPER DAEMON **r258838**

THE CASPER DAEMON, developed by Pawel Jakub Dawidek, builds upon the Capsicum and Capabilities security technologies. Co-sponsored by the FreeBSD Foundation and the Google Open Source Programs Office, casperd(8) provides an easy to use interface to access system services from within a capabilities sandbox.

Capsicum is a sandbox framework that extends on the POSIX API, providing two system call interfaces—capability mode and capabilities. Capability mode restricts processes from accessing global namespaces, such as the file system. Capabilities limit what operations can be performed on file descriptors.

casperd(8) provides host services to sandboxed components that would otherwise be inaccessible. One example of this is DNS resolution, where sandboxed components are prohibited from sending UDP packets to arbitrary destinations. To allow sandboxed components access to DNS resolution, requests are proxied through a dedicated, non-sandboxed process.

Several applications have had Capsicum support added, which can take advantage of the security benefits provided by component sandboxing and interfacing with casperd(8), such as dhclient(8), ping(8), tcpdump(8), and auditdist(8). Capsicum support in additional applications is progressing.

casperd(8) is enabled by default in FreeBSD-CURRENT as of revision **r259581**.

THE VT SYSTEM CONSOLE **r259016**

THE VT SYSTEM console, developed by Aleksandr Rybalko under sponsorship of the FreeBSD Foundation, has been committed to FreeBSD-CURRENT. The vt driver, also known as "Newcons", serves as a replacement for the sc(4) console driver, providing a number of enhancements ranging from UTF-8 font support, to enabling users running X11 with Kernel Mode Setting ("KMS" for short) to switch back to console from X11.

When KMS support was added to FreeBSD for Intel and Radeon HD graphics cards, it would not be possible to switch back to the console once X had started. This is due to the sc(4) driver lacking support for KMS, which in some cases, would require rebooting the system to recover graphical output. The vt driver restores the ability to switch between an X11 session and the console.

VT also brings UTF-8 support to the system console. Currently, Latin and

Cyrillic fonts are supported, and work on support for additional fonts is in progress.

For those interested in testing the new vt console driver, a new (temporary) kernel configuration file "VT" has been added to head/. This kernel configuration file includes the GENERIC kernel, and adds the necessary options to use the new driver in place of the sc(4) driver.

Alternatively, for those interested in testing the new driver on a new installation of FreeBSD, recent snapshot builds have included building separate VT-enabled installation images for the amd64 and i386 architectures. These images can be found at:

http://ftp.FreeBSD.org/pub/FreeBSD/snapshots/ISO-IMAGES/

(The "svn update" column author would also like to note that this column is written on a laptop running FreeBSD-CURRENT using the VT driver.)

DRAGONFLY MAIL AGENT IMPORTED **r262282**

THE DRAGONFLY Mail Agent, DMA, is a lightweight mail transport agent, useful for deployments where it is ideal to have system mail (for example, daily periodic output) delivered, but the system should not receive mail.

In addition to NULLMAILER functionality, DMA supports TLS/SSL and remote SMTP authentication.

32-BIT BHYVE SUPPORT **r261504**

PRIOR TO THIS change, running 32-bit instances of FreeBSD within bhyve was not supported. As result of a number of changes from John Baldwin, it is now possible to run both amd64 and i386 virtual machines within bhyve.

NETMAP UPDATES r261909

IN ADDITION to a number of bug fixes and performance improvements, this update to netmap provides bidirectional blocking I/O while moving over one hundred million packets per second. Additionally, kqueue support has been added, which is needed by bhyve, and segmentation offloading for the VALE switch, which is useful between VMs.

"ONIFCONSOLE" ADDITION

r260913

A NEW FLAG has been added to the ttys(5) file—"onifconsole." The new flag is equivalent to "on" (activating the serial console) when the tty is an active kernel console, otherwise it defaults to "off."

This is particularly useful for embedded systems, where there may be one or more serial channels available, or on systems with IPMI SoL (serial over LAN) connections, and the "default" tty may differ.

As a hobbyist, Glen Barber became heavily involved with the FreeBSD project around 2007. Since then, he has been involved with various functions, and his latest roles have allowed him to focus on systems administration and release engineering in the Project. Glen lives in Pennsylvania, USA.

THE SECOND PORTS report

BY THOMAS ABTHORPE

Ports Report is a summary of recent activity in the Ports infrastructure, news in the world of Ports, tips & tricks.

NEW PORTS COMMITTERS

When you do good work and continually spam other committers with quality PRs, you might be asked to join as a ports committer. Recent additions as committers are: Mikolaj Golub, a FreeBSD source committer who has expanded his role, Bruce Mah, a long time FreeBSD source committer, and former members of FreeBSD Release Engineering team, Alexey Degtyarev, Jonathan Chu, Thomas Zander, Rodrigo Osorio, Michael Gmelin, and Kurt Jaeger. We are pleased to add all this new talent as Ports Committers.

CHANGES IN THE FREEBSD PORTS MANAGEMENT TEAM

In November 2013, the team launched a pilot project called portmgr-lurkers@. The goal of this project was to give porters a chance to see the inner workings of portmgr@, contribute something to the infrastructure in some meaningful way and to form the basis of a pool of candidates that could join the ranks of the team at a later date. As part of the inaugural launch, Mathieu Arnold and Anotine Brodin were the first candidates to take on the task. They rose to the challenge of the job and were made full members of the team in January 2014.
With the addition of two new members, we bid farewell to two members of portmgr@. After many years of service, Joe Marcus Clarke and Ion-Mihai Tetcu stepped down from their duties. Joe was our longest serving member of the team. Among his many accomplishments was being the repocopy source of authority, instrumental in championing tinderbox development and maintaining portlint. Ion-Mihai is the person who initially gave us Ports QAT, which provided almost immediate response to a problematic commit—and then he enhanced with other tests such as installation of all ports to a non-standard DESTDIR. These combined efforts paved the way for cleaner commits to the ports tree.

VALUE-ADDED SERVICES FROM PORTMGR@

One of the behind-the-scenes jobs for portmgr@ is something called portmgr-feedback@. This was the original work of Port Manager Emeritus Mark Linimon and was a way to combine his responsibilities on portmgr@ and bugmeister@. When ports are old, stale, unfetchable, or in some state of disrepair, the maintainer is contacted with a cc: to portmgr-feedback@. Any replies that come from the maintainer are the responsibility of various members of portmgr@ to try to resolve. Most responses are last-ditch efforts of contributors who just want their port updated. In recent months, members of the portmgr-lurkers have earned their stripes working in this capacity.

TIPS FOR PROSPECTIVE PORTERS

Staging all ports is the future of the ports tree. In the old model of the tree, you would compile your port in a scratch folder and then install it to its destination as root. If the port had been improperly prepared or if something went wrong during the install phase, you might have remnants of an incomplete installation. With staging, your port builds in your scratch directory, and then installs into your staged directory, generating a tarball upon completion. It is from this tarball that your port gets installed to its final destination.

The first step is to remove the NO_STAGEDIR=yes from your ports Makefile. The next steps vary based on the build system of your

WHAT'S ON THE HORIZON?

The pkgng support infrastructure has been in the works for a couple of years now and will replace the old pkg_* support. The decision has been made to allow the old pkg_* software to be EoL'd 6 on September 1, 2014 in all active FreeBSD branches.

People are encouraged to start testing pkg(8) in their test environments before taking it live. You will find the benefits of full binary updates for your ports to be useful in a short amount of time. Even if you prefer to compile from source, you will still reap the benefits of the modern packaging system.

You can read more about pkgng on the FreeBSD wiki, https://wiki.freebsd.org/pkgng and in this issue.

port source code. Sometimes it may just be a matter of changing some build targets in your Makefile, other times you might have to customize an install phase. This is where you, as a porter, have to use your imagination. The great thing is that a lot of people have already staged their ports and you already have some good examples in the tree.

One of the monotonous tasks of maintaining ports is the plist generation. If you have your port staged correctly, you can simply issue the command "make makepllist" and the infrastructure will generate the list to stdout, you can redirect this output to pkg-plist and your work is almost done. (You have done full testing on your port right?:-D)

If your port relies on cmake to build, you are more fortunate than you can imagine. You do not have to imbed any additional directives in your Makefile for staging, the infrastructure just takes care of it for you, and you just have to capture your plist.

You can read more about staging at https://wiki.freebsd.org/ports/StageDir

DOING YOUR PART TO IMPROVE THE PORTS TREE

At any given time, approximately 20% to 25% of ALL ports are unmaintained. Orphaned ports need updates too! If you have one installed on your system, consider adopting it. The following shell script will generate a list of ports you have installed on your system that do not have a maintainer

```
cd /usr/ports; grep -F "`for o in \`pkg_info -qao\` ; \
do echo "|/usr/ports/${o}|" ; done`" `make -V INDEXFILE` | \
grep -i \|ports@freebsd.org\| | cut -f 2 -d \|
```

Once you have identified some prospective ports, do your research to see if the port needs updating and file a Problem Report to update and claim ownership of the port.

You can read more about adopting ports at, http://www.freebsd.org/doc/en/articles/contributing-ports/adopt-port.html.

The ports tree offers you a lot of software that gives you advanced usability, but it does not happen by itself. Iit is a user driven effort, so get involved in some way, whether it is adopting a port, or offering patches for other ports.

http://fb.me/portmgr —*"Like" us*
http://twitter.com/freebsd_portmgr — *Follow us*
http://blogs.freebsdish.org/portmgr/ — *Our blog*
https://plus.google.com/u/0/communities/108335846196454338383 — *G+1 us*

Thomas Abthorpe is a server administrator with over 20 years in the industry. He got his Ports commit bit August 2007, joined the Ports Management Team in March 2010, and was elected to FreeBSD Core Team in July 2012. When he is not busy doing FreeBSD business, he volunteers as an apprentice bicycle mechanic with Bicycles for Humanity.

BSDCAN 2014

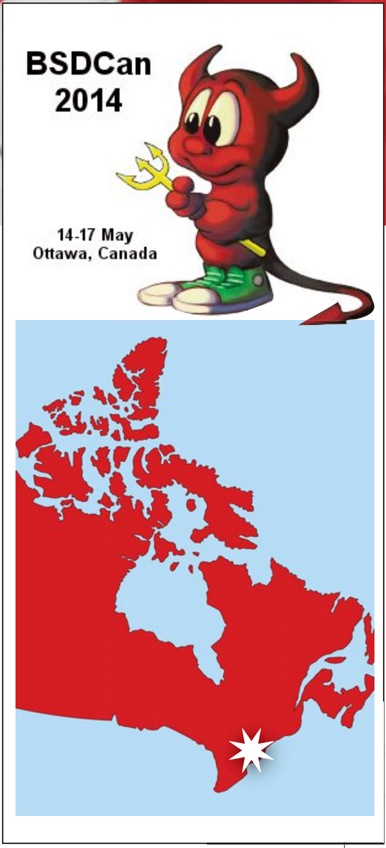
The 11th Annual BSDCan!

THE TECHNICAL BSD CONFERENCE. High Value, Low Cost, Something for Everyone! BSDCan, a BSD conference held in Ottawa, Canada, has quickly established itself as the technical conference for people working on and with 4.4BSD based operating systems and related projects. The organizers have found a fantastic formula that appeals to a wide range of people from extreme novices to advanced developers.

WHERE ....Ottawa, Canada
University of Ottawa

WHEN...................Wed. & Thurs. May 14-15 (tutorials)
Fri. & Sat. May 16-17 (conference)

GO ONLINE FOR MORE DETAILS
http://www.bsdcan.org



• April, May and June

Events Calendar

BY DRU LAVIGNE

These BSD-related conferences are scheduled for the second quarter of 2014.

More information about these events, as well as local user group meetings, can be found at **bsdevents.org.**

GRAZER LINUXTAGE • April 4–5, 2014 Graz, Austria
http://linuxtage.at/ • This community-organized conference is free to attend. There will be a BSD booth in the expo area and the BSDA certification exam will be available. Typically there is at least one BSD-based presentation.

LINUXFEST NORTHWEST • April 26–27, 2014 Bellingham, WA
http://linuxfestnorthwest.org/ • This is the 15th year for this annual, community-based conference. This event is free to attend and always has at least one BSD booth in the expo area. This year's presentations include talks on FreeNAS and on ZFS.

LINUXTAG • May 8–10, 2014 Berlin, Germany
http://linuxtag.org/2014/ • LinuxTag is Europe's leading conference and exhibition for professional users and developers of open source. There will be several BSD booths in the expo area and there are typically several BSD-specific presentations.

BSDCAN • May 14–17, 2014 Ottawa, Canada
http://www.bsdcan.org/2014/ • The 11th annual BSDCan will take place in Ottawa, Canada. This popular conference appeals to a wide range of people from extreme novices to advanced developers of BSD operating systems. The conference includes a Developer Summit, Vendor Summit, Doc Sprints, Tutorials, and presentations. The BSDA certification exam will be available during the lunch break on May 16 and 17 and the first beta of the BSDP lab exam will launch on May 18.

TEXAS LINUXFEST • June 13–14, 2014 Austin, TX
http://texaslinuxfest.org/ • Texas Linux Fest is the first statewide, annual, community-run conference for Linux and open source software users and enthusiasts from around the Lone Star State. There will be a BSD booth in the expo area and several BSD-themed presentations.

SOUTHEAST LINUXFEST • June 20–22, 2014 Charlotte, NC
http://www.southeastlinuxfest.org/ • This is the sixth edition for this open source conference in the Southeast. There will be a BSD booth in the expo area and several BSD-related presentations.

CALENDAR ANNOUNCEMENT

By Wojciech A. Koszek

FreeBSD in the 10th Annual Google Summer of Code 2014

Everybody knows Google, but for some, Google Summer of Code (GSOC) may be a new concept. For those who don't already know about it, the event began in 2005 with the goal of attracting more engineering talent to the open source world. Google Summer of Code is a global program that offers students stipends to write code for open source projects. The program has worked with the open source community to identify and fund exciting projects so that qualified students with an interest in and passion for computers and engineering can spend their summer writing open source software. The idea behind the event is straightforward: To match open source projects with students and have them work together to improve free software and create more engineering spark in the software community.

During the 12 weeks of the event, mentors from organizations accepted by Google will review students' ideas and help with planning, implementation, and scoring projects. Upon completion, some students can receive prizes up to $5,500.

March 10th is the start of student applications. This year is the event's 10th anniversary, and we're pleased to announce that for the 10th time The FreeBSD Project has been officially accepted to Summer of Code. The FreeBSD Project has already received several queries from students who are interested in improving our software stack and we're very excited about it. However, GSOC requires engagement from the entire community if it is to be a complete success.

You can help. This year we are advertising FreeBSD and the Google Summer of Code together. We plan to provide universities around the world with a short introduction to GSOC/FreeBSD along with promotional materials and pointers on how to apply.

You can suggest an idea for an interesting project, help mentor students, or simply spread the word in your local community. It all helps and will hopefully attract the future leaders of The FreeBSD Project.

Don't miss out on this opportunity to make FreeBSD a better system.

March 10th is the start of student applications

APPLY	http://gsoc.freebsd.org/
SUGGEST ideas	http://tinyurl.com/FreeBSD-GSOC2014
PRINT a poster	http://tinyurl.com/FreeBSD-GSOC2014-poster

FreeBSD™ JOURNAL

LEAK THIS!

● BLOG IT ● TWEET IT ● READ IT

FreeBSD Journal is available as a mobile app for iPad, iPhone, Kindle, and all Android devices at $19.99 per year (for 6 issues), or $6.99 per single copy.

FreeBSD™ Journal is published by The FreeBSD Foundation

this month In FreeBSD

BY DRU LAVIGNE

As I peruse the April calendar for inspiration, I note that April Fool's Day is on the 1st and Easter (both Western and Orthodox) is on the 20th. Naturally, my geek mind thinks of RFCs and Easter eggs, the subjects of this issue's column.

Speaking of calendars, did you know that your FreeBSD system comes with a built-in calendar? Try typing *cal* or *ncal* from the command line to see this month's calendar. While handy as is, the built-in calendar is far nerdier than that. I know that both Western and Orthodox churches are celebrating Easter on the same day this year, as I typed *ncal -e* (for Eastern) and *ncal -o* (for Orthodox). Usually they're not. Try *ncal -e 2000* and *ncal -o 2000* for an example.

In addition to Easter, you can view the calendar for any specified month and year. It's trivial to find out which day of the week Neil Armstrong walked on the moon (*cal july 1969*) or the day of the week Captain Cook died (*ncal 2 1779*). The most interesting month occurred when the calendar switched from Julian to Gregorian. Try typing *cal 9 1752* to see what I mean.

Easter Eggs

According to Wikipedia, "an Easter egg is an intentional inside joke, hidden message, or feature in a work such as a computer program." FreeBSD has a few Easter eggs tucked away in its commands and source code.

As an example, try typing make love on a FreeBSD 9.x system and on a 10.x system. While both answers are as cheezy as the command, there is an open PR (www.freebsd.org/cgi/query-pr.cgi?pr=179833) to return to the previous behavior. On an equally cheezy note, one could ponder the difference between man man and man woman.

Some Easter eggs are built into the name of a command. For example, biff, the mail notification program, was named after a dog who attended lectures with his owner in Evans Hall at Berkeley. There are conflicting reports on whether or not he actually liked to bark at the mailman. However, cat is not named after anyone's cat. The commands more and less are also named in an Easter egg fashion, seeing that more is less than less and less is more than more.

Some Easter eggs are hidden as functions within the source. A famous example can be seen in grep gravy /usr/src/sbin/shutdown/shutdown.c, which is a reasonable function name for those times when a system just won't listen to reason. Some Easter eggs aren't always considered politically correct, something that Jordan Hubbard discovered when he left FreeBSD for Apple. Apparently, a grep for DOOFUS caused a bit of a stir, though it does remain in FreeBSD. You can read about the commotion at https://groups.google.com/forum/?hl=en#!msg/fa.freebsd.hackers/V7cWmQeWVfg/IljCCbdz4BEJ.

Some Easter eggs are innocuous, only having special meaning to the developer until their meaning is deciphered and disseminated. An example is seen in Kirk McKusick's Wikipedia page: "the magic number used in the UFS2 super block structure reflects McKusick's birth date: #define FS_UFS2_MAGIC 0x19540119 (as found in /usr/include/ufs/ffs/fs.h)". Now that you know, don't forget to send him a birthday greeting next year.

Do you know of, or have you created, any other FreeBSD Easter eggs?

What's your favorite RFC?

Send your comments to feedback@freebsdjournal.com.

RFCs

Moving on to April Fool's Day, those of us who tend toward networking look forward each year to see if there will be an RFC (Request for Comments) published on April 1. Even if RFCs aren't what you would consider casual reading for anyone other than an ardent insomniac, you can still appreciate the humor in the April 1 RFCs. If you ever wondered if the legends of the Internet have a sense of humor, check out http://www.apps.ietf.org/rfc/apr1list.html for such gems as the ARPAWOCKY, Twas the Night Before Start-up, The Twelve Networking Truths, and the Etymology of Foo. The Bergen Linux and BSD User Group wrote about their implementation of RFC 1149 at http://www.blug.linux.no/rfc1149/.

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with BSD CERTIFICATION

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification **YOU'LL HAVE WHAT IT TAKES!**

SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration